

第163回お試しアカウント付き講習会  
「スーパーコンピュータ—超入門」  
2021年9月30日

---

本資料はWindowsマシン上で  
**PuTTY**を使われる方向けのものです

# 本講習会の内容

13:00 – 14:00

注意事項、Zoomの使い方

スーパーコンピューターについて初歩(講義)

スーパーコンピューターへのログイン(講義・実習)

14:15 – 16:00

プログラムの作成・コンパイル・実行(講義・実習)

並列プログラムのコンパイル・実行(講義・実習)

16:15 – 17:00

より進んだ利用に向けて(講義)

## 講習会の注意事項

- 1か月利用可能なアカウントが配布されます。
  - 利用規程に記載の用途以外に使用しないでください。
    - ✓ 研究、教育、社会貢献のために使用
  - 1か月经過後(10/30)にはアカウントが削除されるため、ファイルも削除されます。
  - アカウントの継続利用のためには、講習会を最後まで受講頂く必要があります。
- 企業の方が当たりセンタートライアルユース制度に申し込む場合、センターが開催するいずれかの講習会を修了してください。（本日の講習会を想定する場合、最後まで出席のこと）
- 本講習会に関する質問は  
[shiba@cc.u-tokyo.ac.jp](mailto:shiba@cc.u-tokyo.ac.jp)  
に送付ください。センターのスパコン相談窓口等には送付しないようご注意ください。

# Zoom

## ■ 講義・実習中の質問方法

- 「手をあげる」機能
  - 質問がある際、全体の状況を確認するため使用
- ブレークアウトセッション
  - 画面を共有しながらエラー対応する際に使用
  - (なるべく口頭でのやりとりで対応する予定)

- [https://utelecon.adm.u-tokyo.ac.jp/zoom/how\\_to\\_use](https://utelecon.adm.u-tokyo.ac.jp/zoom/how_to_use)

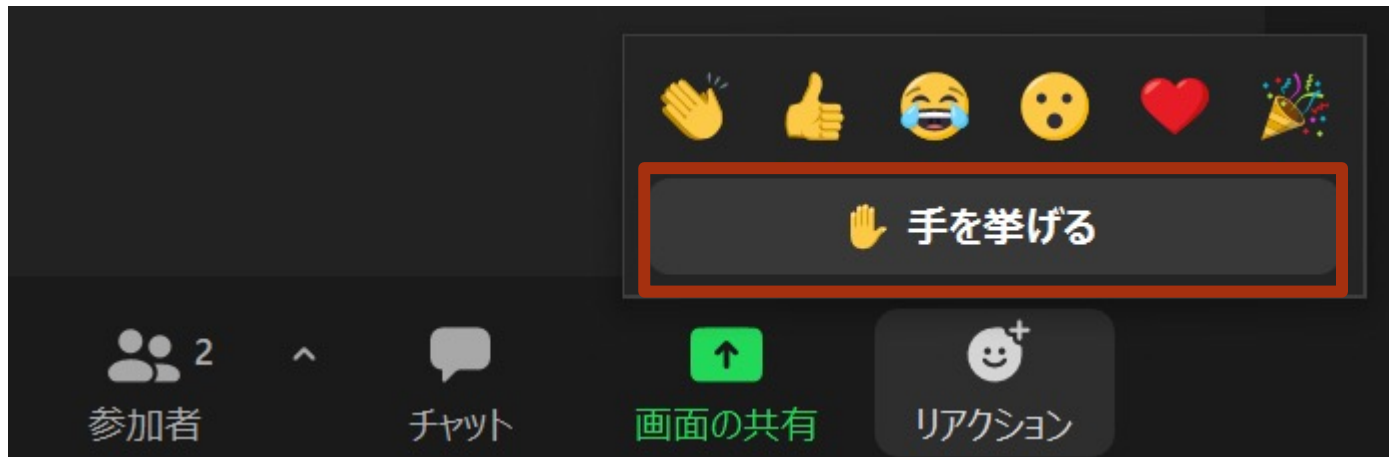


# Zoomで手を挙げる方法

1. Zoomメニュー中の「リアクション」をクリック

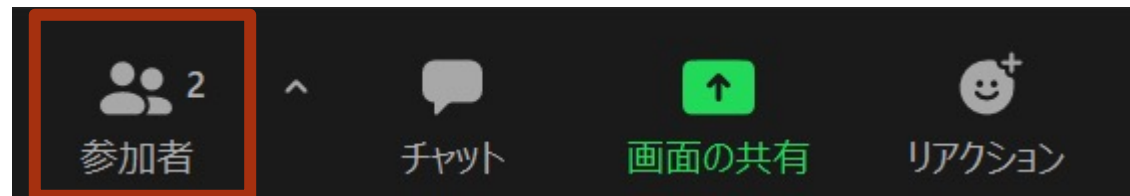


2. ポップアップで表示された「手を挙げる」をクリック

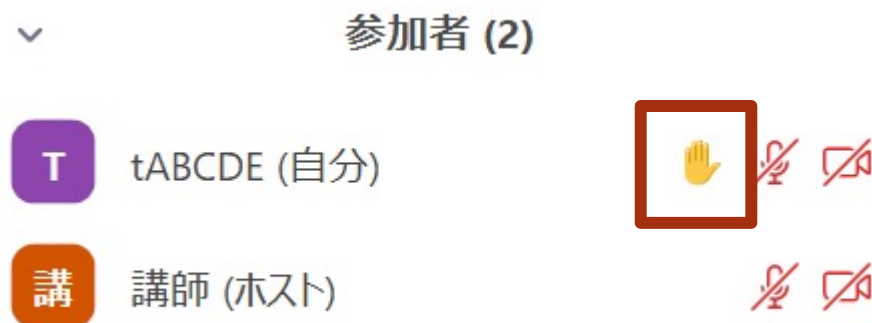


# Zoomで手が拳がっていることの確認方法

1. Zoomメニュー中の「参加者」をクリックして、参加者一覧を表示

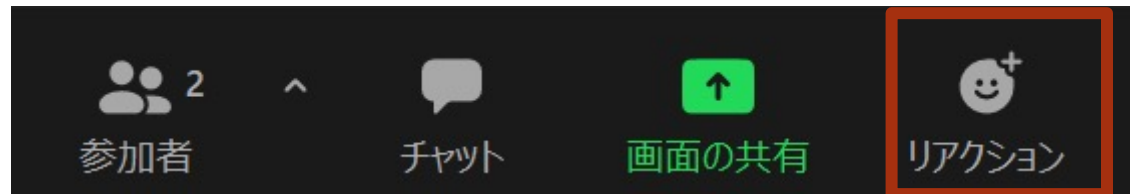


2. 表示された参加者一覧の、自分のところを見ると手が拳がっている

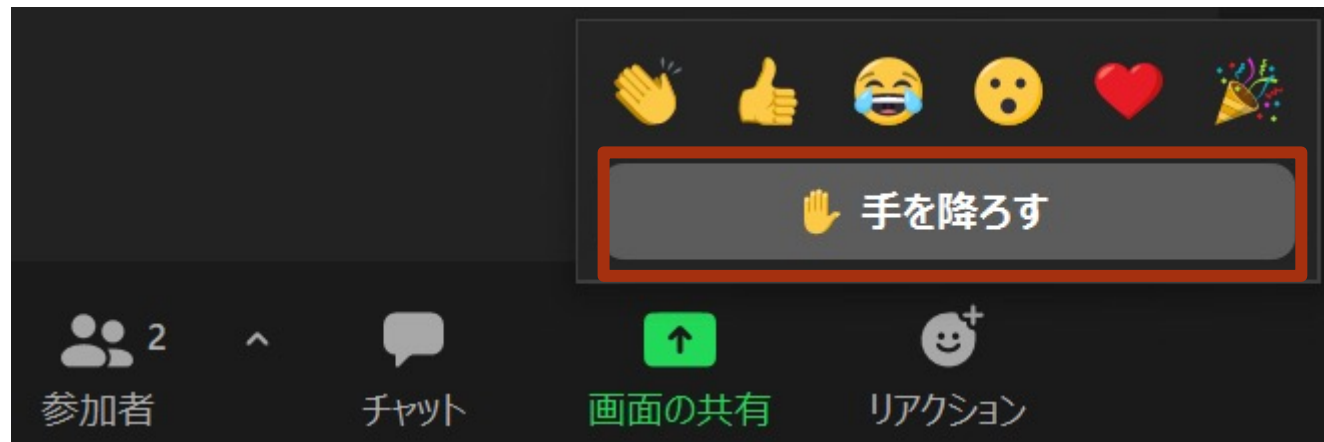


# Zoomで手を下ろす方法

1. Zoomメニュー中の「リアクション」をクリック



2. ポップアップで表示された「手を降ろす」をクリック



## ブレイクアウトルーム（1/6）

- 演習時に使用するかもしれません
- 演習中に「ヘルプを求める」ことができます
  - ホストを招待した後に「画面を共有」することで、皆さんの記述したプログラムを一緒に見ながら問題解決にあたります
- Zoomメニュー中の「ブレイクアウトルーム」をクリック



## ブレイクアウトルーム (2/6)

- ・ 進行中のブレイクアウトルームのリストが表示されるので、空いている部屋に「参加」してください
  - ・ 左の例では5部屋がすべて空室、右の例ではルーム1のみ参加者がいる



ブレイクアウトルーム- 進行中

▼ ルーム1	参加
▼ ルーム2	参加
▼ ルーム3	参加
▼ ルーム4	参加
▼ ルーム5	参加

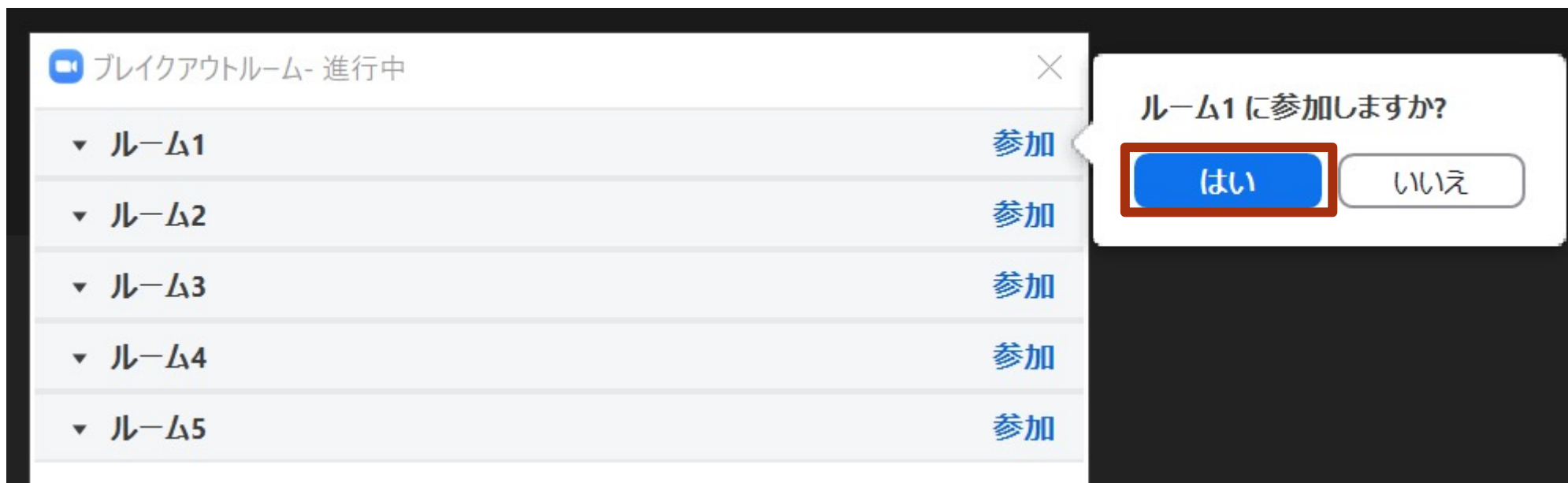


ブレイクアウトルーム- 進行中

▼ ルーム1	参加
● tABCDE	
▼ ルーム2	参加
▼ ルーム3	参加
▼ ルーム4	参加
▼ ルーム5	参加

## ブレイクアウトルーム (3/6)

「参加」をクリックすると確認画面が出てくるので、「はい」を選択すると入室できます



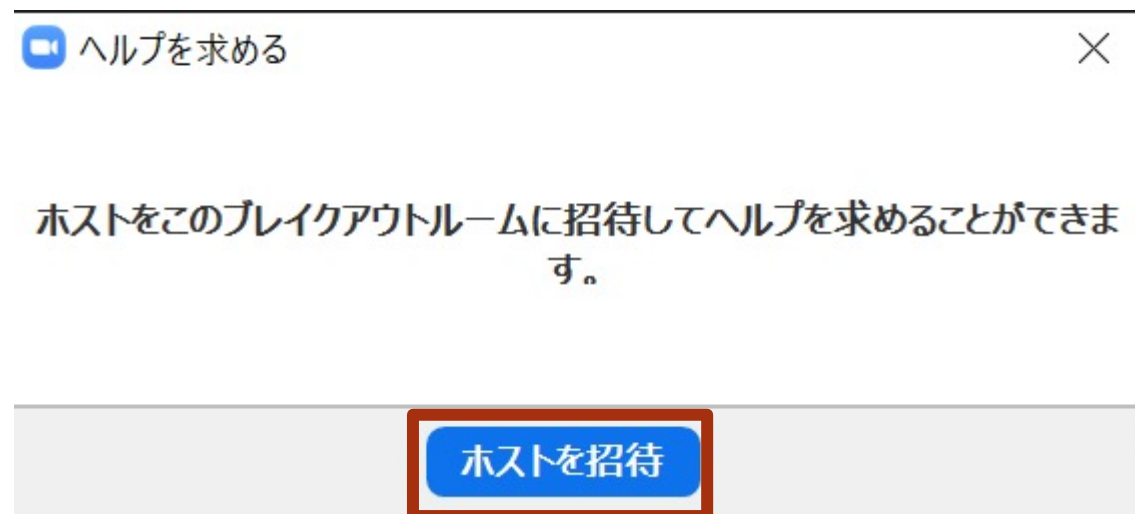
## ブレイクアウトルーム（4/6）

再度メニュー中の「ブレイクアウトルーム」をクリックすると、「ヘルプを求める」が増えているのでクリックしてください



## ブレイクアウトルーム（5/6）

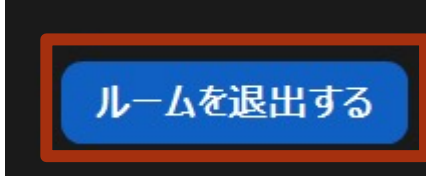
ポップアップで出てきた「ホストを招待」をクリック  
ホスト（講師）の参加待ちに移行します（画面はそのまま）  
他の受講者のヘルプ中など、直ちに対応できない場合もあります





## ブレイクアウトルーム（6/6）

問題解決後は、Zoomメニュー中の「ルームを退出する」



ルームを退出する

「ブレイクアウトルームを退出」が表示されるのでクリックして、元の講習会会場にお戻りください

間違えて「ミーティングを退出」すると講習会から退出します



ミーティングを退出

ブレイクアウトルームを退出

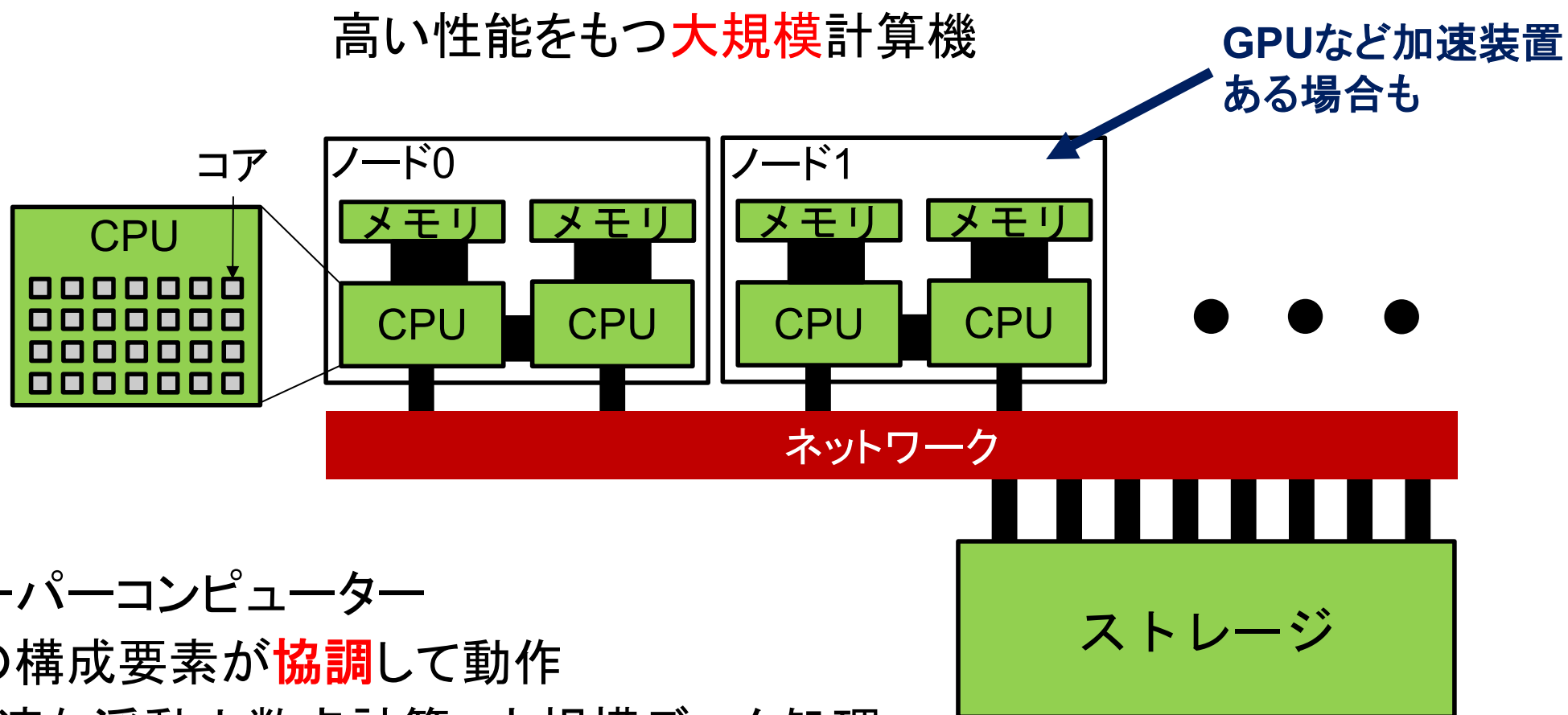
キャンセル

# 東大スーパーコンピューター OBCX

Oakbridge-CX @ 東京大学柏キャンパス (2019-)



# スーパーコンピューター、略してスパコン



ここ20年のスーパーコンピューター

= **多数**の構成要素が**協調**して動作

→ 高速な浮動小数点計算、大規模データ処理

高性能なノード、高速なネットワーク、大容量かつ高速なストレージ

# 性能指標 ～ OBCXを例に (1)

多数のコア、高いメモリ転送速度

		iMac (2018)	OBCX
CPU	コア数	6	56 (2ソケット)
	周波数 [GHz]	3.0	2.7
	理論演算性能[TFlops]	0.58	4.84
Memory	容量 [GByte]	16	192
	転送速度 [GByte/秒]	42.6	281.6



580 mm

ボードには水冷  
(とにかく熱い)!

174.3 mm

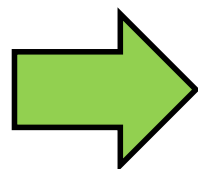


# 大きいことはいいこと (もある)

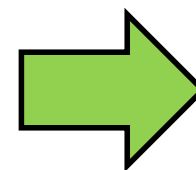
たくさんのパソコンを同時に連携して使用するようなもの



× 4



× 17



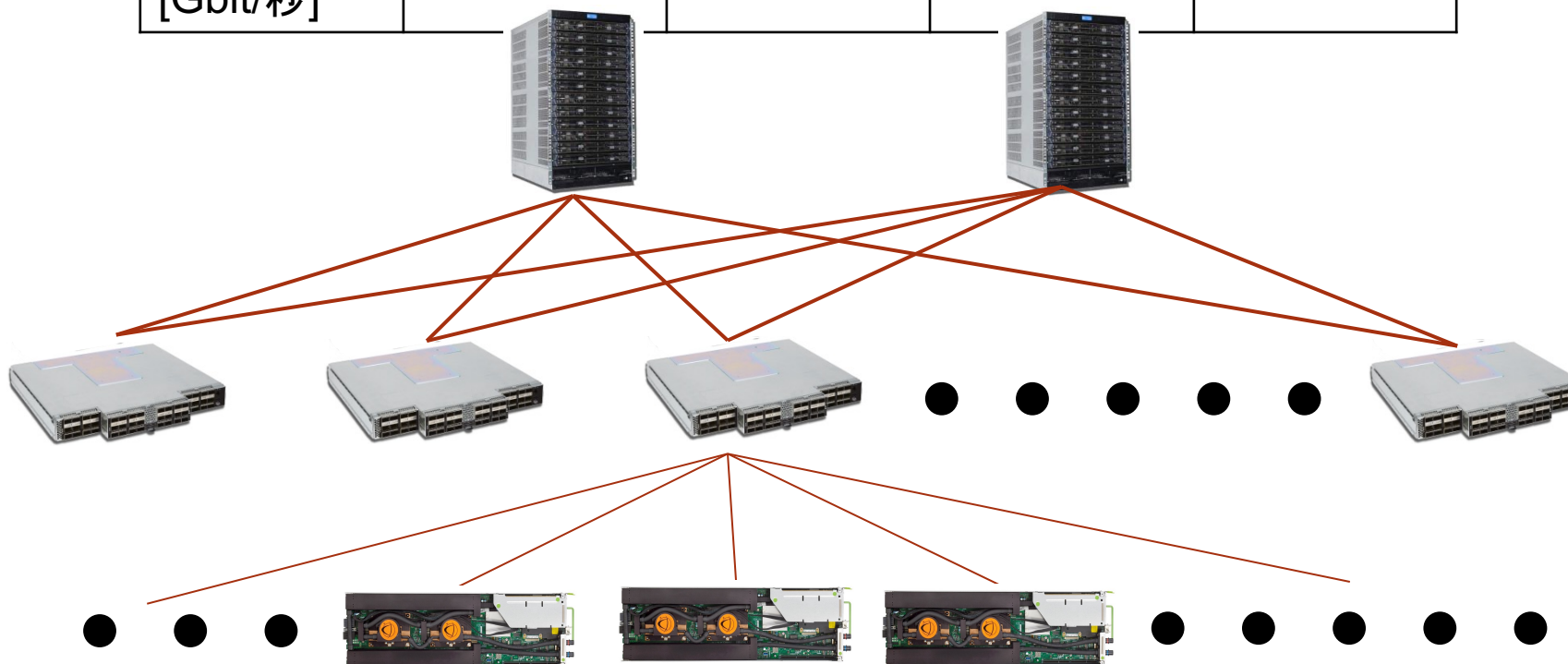
× 21

全部で1,368台

# 性能指標 ～ OBCXを例に (2)

多数のノード数を全体で繋ぐためにネットワークは非常に高速

	携帯 4G	無線 (Wifi 11ac)	有線 (Gigabit Ether)	Omni-path (OBCX)
転送速度 [Gbit/秒]	0.1 ~ 1.0	6.9	1	100



## 性能指標 ～ OBCXを例に (3)



### 大容量・高速のストレージ

OBCX Lustre storage

- 容量 : 12.4PByte (12400TByte)
- 構成 : HDD 8TB x 1,360本
- 転送速度 : 193.9GByte/秒

HDD,SSDの転送速度は  
HDD : 0.1～0.2GByte/秒  
SSD : 4～5GByte/秒

\*2020年9月現在



# OBCX ハードウェア仕様

## 計算ノード

Chassis: PRIMERGY CX400 M4 x342 <4node / Chassis>  
 Node: PRIMERGY CX2550 M5 x1,240, CX2560 M5 x128



x1,368 node



### 全体性能

理論演算性能: 6.61PF  
 主記憶容量: 256.5TiB  
 メモリバンド幅: 385.1TB/s  
 ラック数: 21ラック  
 SSD搭載: 128ノード

### ノード単体

理論演算性能: 4.8384 TF  
 手記憶容量: 192GiB  
 メモリバンド幅: 281.6GB/s

## 計算ノード間ネットワーク (Omni-Path Architecture) 通信性能 100Gbps

### ログインノード



x10

FUJITSU Server  
 PRIMERGY CX2560 M5 x 10

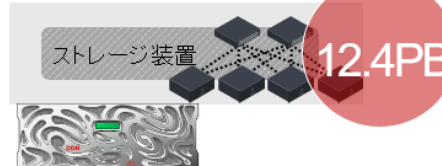
### 管理サーバ群



x15

FUJITSU Server  
 PRIMERGY RX2530 M4 x 15  
 (ジョブ、運用、認証、Web、  
 セキュリティログ保存)

### 並列ファイルシステム



12.4PB

ストレージ装置: DDN ES18KE x2セット  
 ファイルシステム: DDN ExaScaler  
 (Lustreベースファイルシステム)



# 全体性能

FLOPS値 (Floating Point Operations per Second, 浮動小数点演算)

1秒当たりの実数演算性能

- $10^6$  FLOPS = 1 Mega FLOPS = 1 MFLOPS (百万)
- $10^9$  FLOPS = 1 Giga FLOPS = 1 GFLOPS (十億)
- $10^{12}$  FLOPS = 1 Tera FLOPS = 1 TFLOPS (兆)
- $10^{15}$  FLOPS = 1 Peta FLOPS = 1 PFLOPS (千兆)
- $10^{18}$  FLOPS = 1 Exa FLOPS = 1 EFLOPS (百京)

## OBCX

– Intel Xeon Platinum 8280, 1コアの性能は86.4GFLOPS

» 1秒間に864億回の倍精度実数演算

– 1ノード : 56コア

» 4,838.4 GFLOPS = 4.838.TFLOPS ~ 4兆8,384億回

– 全システム : 1,368ノード

**6.618 PFLOPS** ~ 6,618兆3,840億回

# スパコン利用に向けて何を学ぶのか？

Case A ) 新しい計算向けのプログラムを作る  
誰かが作ったプログラムを速くする

特有の技術必要！

繰り返し計算を高速化したい

```
do i=1, 1000000
    c(i) = a(j) * x(i,j) + b(i)
enddo
```

基本のところは C/C++ と FORTRAN で共通

1. CPUが動作しやすい形にプログラムを書き換える方法  
SIMD演算、キャッシュの活用、メモリアクセス改善

2. 協調動作のための分散処理方法、CPU間など通信方法  
OpenMP MPI

3. 上記1. 2. を最適にして演算を行うライブラリの呼出方法

# スパコン利用に向けて何を学ぶのか？

## Case B ) スパコン向けに作られたソフトウェアを利用する

数値流体力学、構造解析、統計解析、  
分子動力学、電子状態計算、量子化学計算、...

ソフトウェアがどのように協調動作するのかの概略を  
理解することにつとめる

→ 並列プログラミングをしない人でも、使ってみた  
速さなどから、計算の仕組みのイメージを獲得

# スパコン利用に向けて何を学ぶのか？

## Case C ) 新しいタイプのデータ処理（学習）をしたい

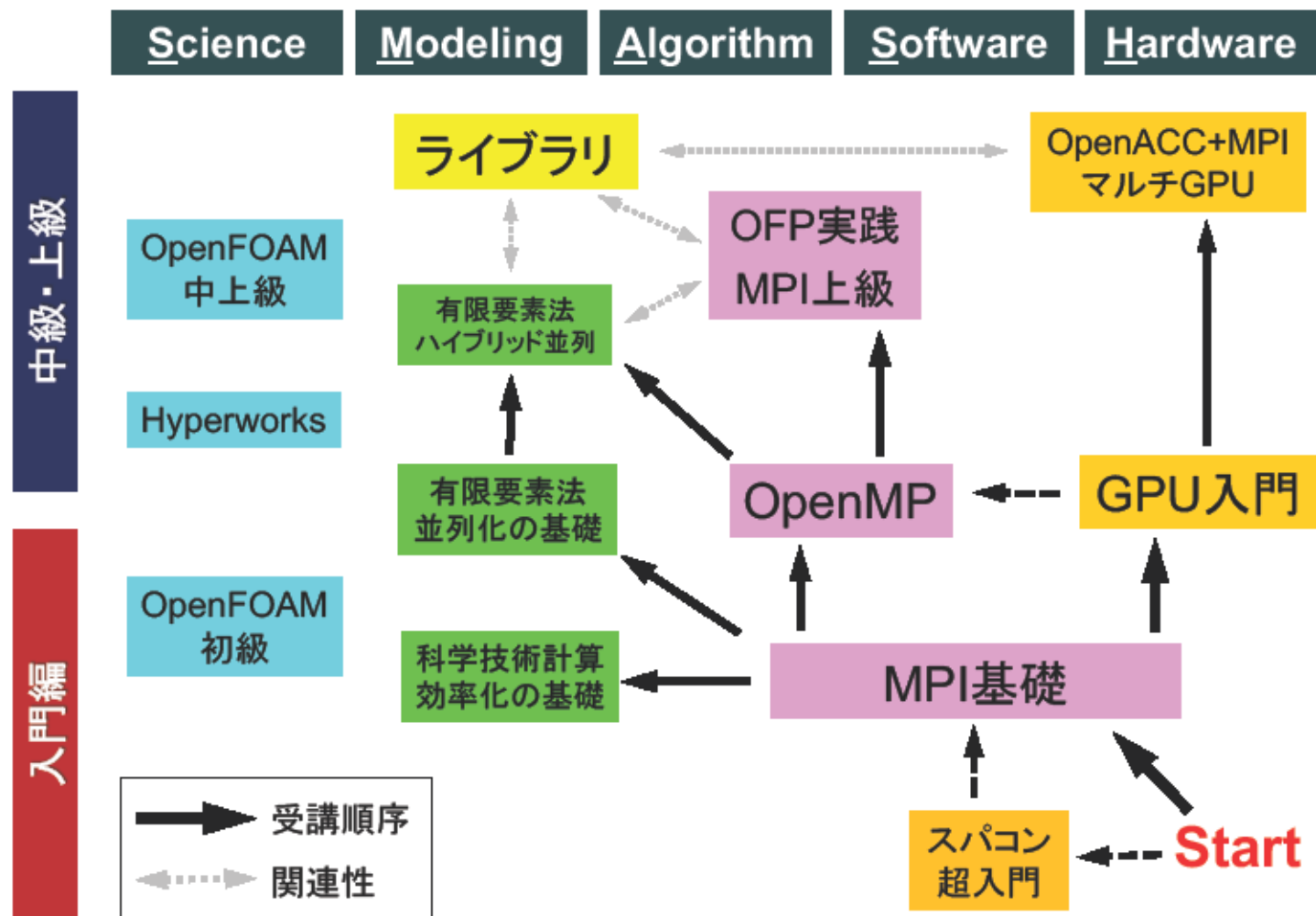
— 多くの方は、手元のマシンで何かされてる方かもしれません

1. Python等による高速なデータ処理
2. スパコン特有の環境構築の方法（コンテナ環境の利用など）  
多数のGPUの使用方法
3. バッチジョブシステム特有の事項

注：大学設置スパコンだとどうしても（民間のクラウドと比較すると）  
少し環境準備が手間がかかるところはあります。

# スパコン向け最適化に関する講習会 @ 東大ITC

目的に応じて受講をご検討ください。



# スーパーコンピュータ (スパコン) 利用のイメージ

通常、スパコンでは  
ログインノードと呼ば  
れる玄関口から実行  
命令を出します



端末

1. ログイン  
SSH

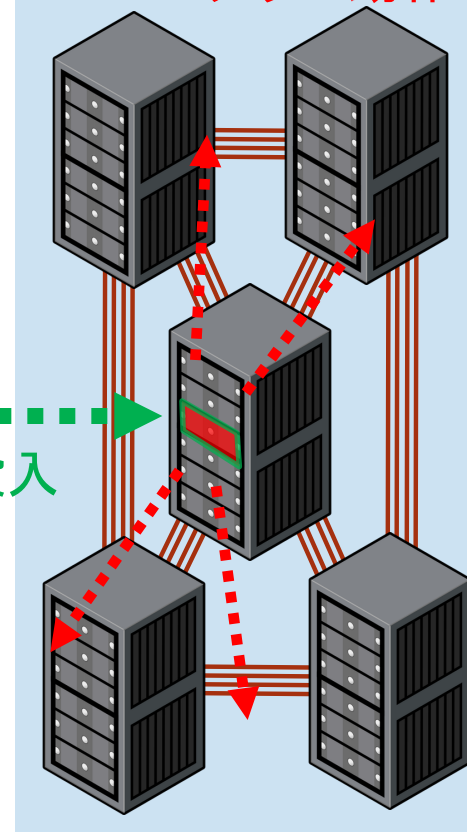


ログインノード

2. ジョブ投入

```
[tut138@obcx02~]$ ls -l  
drwxr-x--- 2 shiba group 10 1  
Apr 13:00 test.out  
[tut138@obcx02~]$ ./test.out  
Hello world  
[tut138@obcx02~]$ qsub a.sh
```

3. プログラム動作



計算ノード  
= 本体

SSH で接続するには  
鍵の準備が必要！

# Secure Shell プロトコル

## 通信が暗号化されたShell

- ShellはOSとユーザーの仲介をする  
コマンドベースのソフトウェア
- 通信データを暗号化し、リモートマシンに  
アクセスできる通信方法:SSH



暗号化された通信を使用して、  
様々なことが可能

- ファイルのコピー
- グラフィカル画面の転送
- トンネリング
- ディレクトリのマウント

### ログイン後の画面の一例

```
[ tUVXYZ @obcx05 ~]$ pwd
/home/ tUVXYZ
[ tUVXYZ @obcx05 ~]$ cd /work/gt00/z30113
[ tUVXYZ @obcx05 tUVXYZ ]$ cd ../
[ tUVXYZ @obcx05 gt00]$ pwd
/work/gt00
[ tUVXYZ @obcx05 gt00]$ cd ~/
[ tUVXYZ @obcx05 ~]$ pwd
/home/z30113
[ tUVXYZ @obcx05 ~]$ cd /work/gt00/z30113
[ tUVXYZ @obcx05 tUVXYZ ]$ mkdir test
[ tUVXYZ @obcx05 tUVXYZ ]$ ls
test
[ tUVXYZ @obcx05 tUVXYZ ]$
```

# 鍵認証方式

より安全な接続をする→鍵認証方式

- パスワードではなく、鍵ペアを使用してログインする方法
- 秘密鍵にもパスワードを設定可能

初期設定(初回ログイン時のみ)

- 鍵ペアを作成
- 公開鍵をログインノードに登録





# 鍵認証方式の注意点

## 注意点

### ■ 秘密鍵の取り扱いに注意

- 厳重に管理してください。

- ✓ 漏洩すると容易にログインできてしまうため。

- 秘密鍵は他のところにコピーしたりしないでください。

- 秘密鍵の入ったPCの紛失などがあった場合は速やかに公開鍵を更新してください。

### ■ 鍵の生成時には必ずパスワードを設定してください。

## よく間違える点

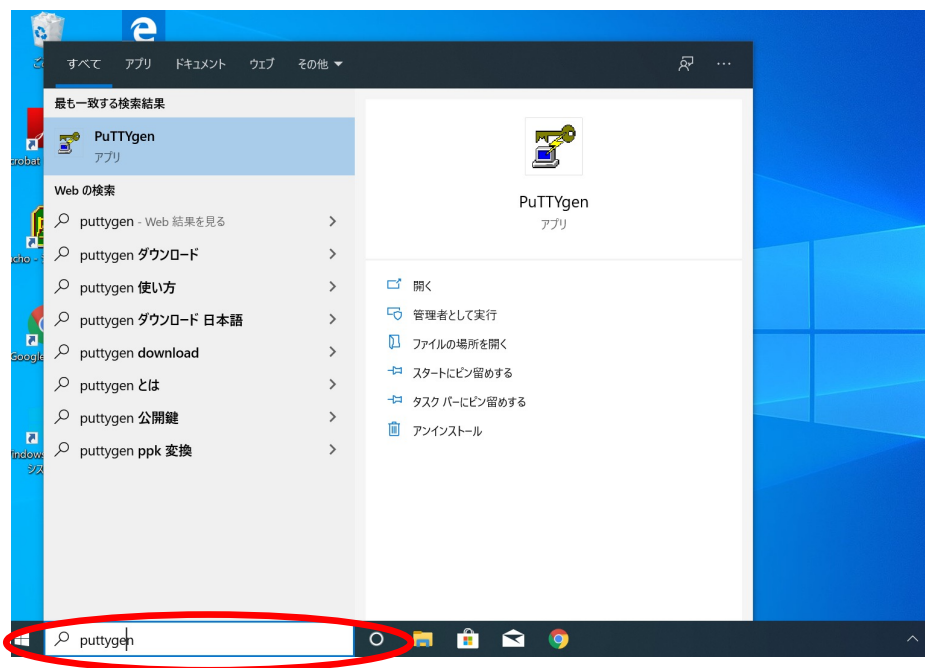
- 秘密鍵のパスワードはOBCXのポータルログインパスワードやログイン後のアカウントのパスワードとは異なります。

# 鍵の作成 (PuTTYの方)

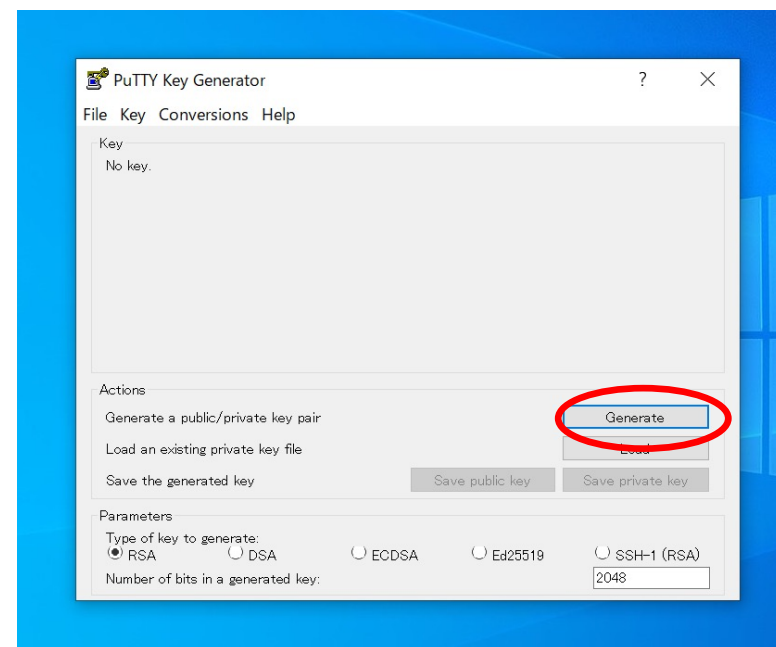
PuTTYgen を用いて公開鍵、秘密鍵を作成します

## (1) PuTTYgen を起動

アイコンがなければ左下のメニューから検索



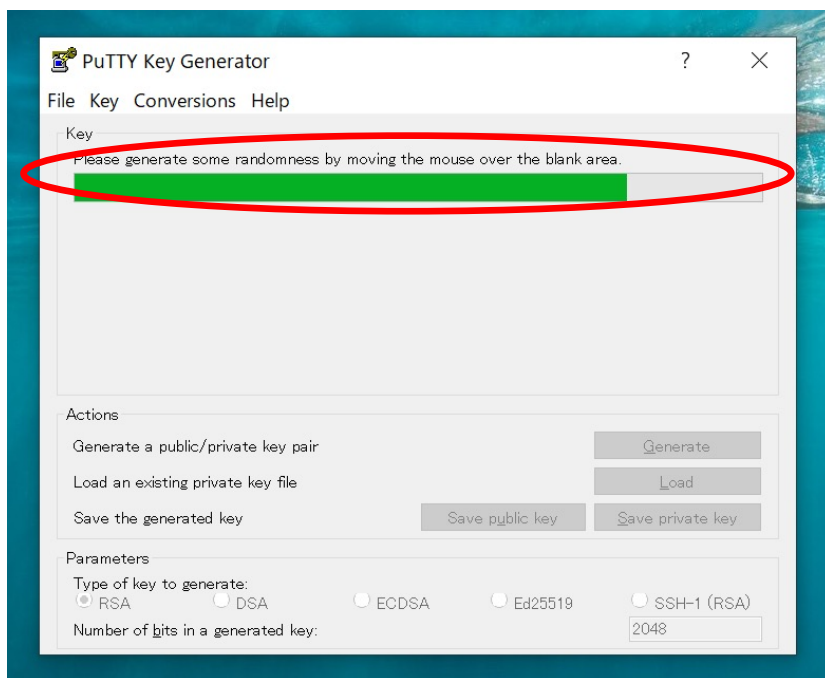
## (2) Generate a public/private Key をクリック



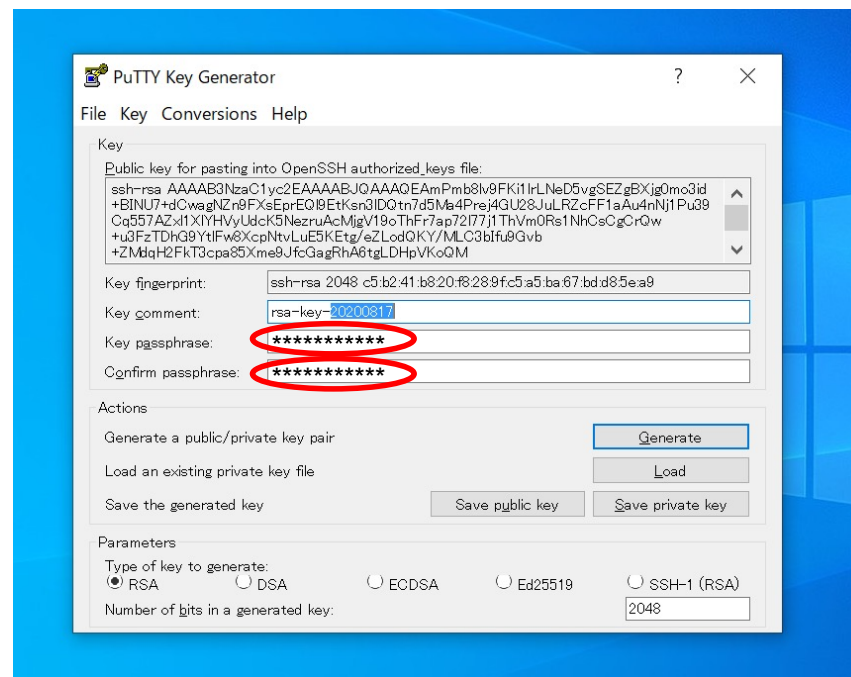
# 鍵の作成 (PuTTYの方)

PuTTYgen を用いて公開鍵、秘密鍵を作成します

(3) 緑色のバーの近くで  
マウスを動かす



(4) 秘密鍵のパスワードを決める

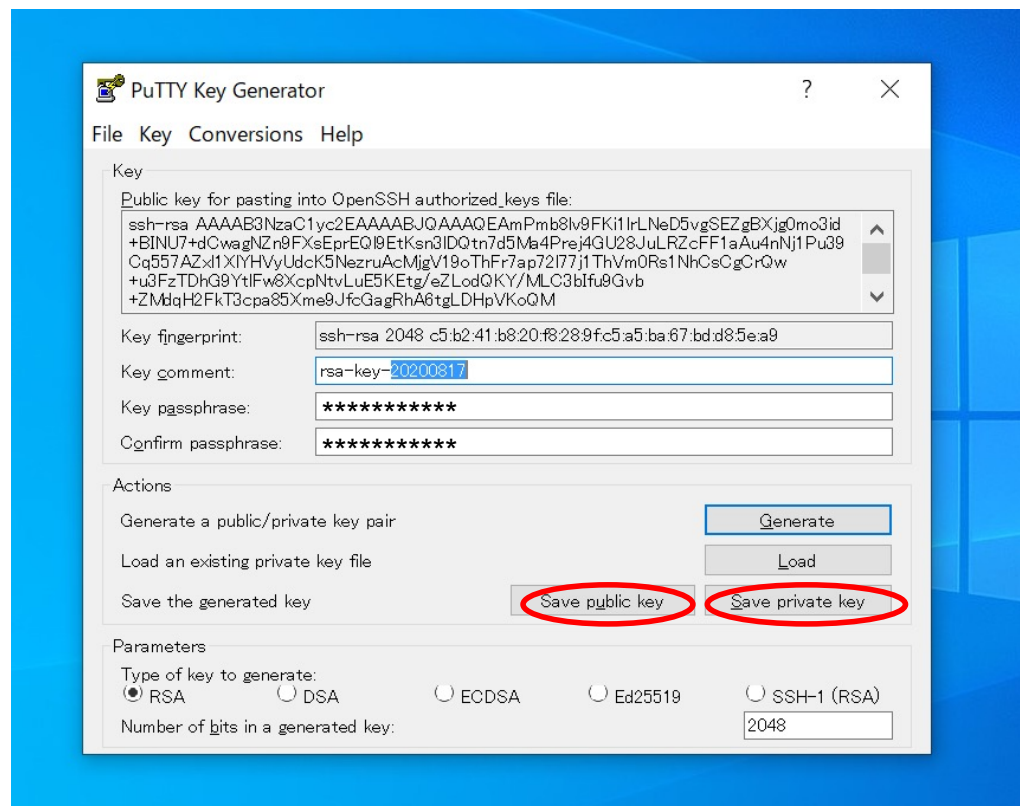


公開鍵、秘密鍵が作成されました

# 鍵の作成 (PuTTYの方)

PuTTYgen を用いて公開鍵、秘密鍵を作成します

(5) 公開鍵と秘密鍵を保存 + 公開鍵をクリップボードにコピー



保存先ディレクトリ情報を忘れずにメモ。あとで使います。

# SSH 公開鍵認証

## id\_rsa

- Private Key（秘密鍵）：PC上
- 文字通り「秘密」にしておく  
作成した場所からコピー・移動しない、他の人に送らない

## id\_rsa.pub

- Public Key（公開鍵）：スパコン上
- コピー可能，他の人にe-mailで送ることも可能
- **もし複数のPCからスパコンにログインする場合は，各PCごとに「公開鍵・秘密鍵」のペアをssh-keygenによって作成**
- **各スパコンには、複数の公開鍵を登録できます**

スパコン上の公開鍵のうちの一つがPC上の「秘密鍵+Passphrase」とマッチすると確認されるとログインできる

# ①スパコンポータルサイトにログイン

<https://obcx-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal.ja/index.cgi>

The screenshot shows a web browser window displaying the Oakbridge-CX portal. The page title is "Oakbridge-CX 利用支援ポータル". The URL in the address bar is "https://obcx-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal.ja/index.cgi". The page content includes a language selector "[ English/Japanese ]" and a "ログイン" (Login) section. The login form has two input fields: "ユーザー名:" (Username) and "パスワード:" (Password). The "ユーザー名:" field is highlighted with a red box, and the "パスワード:" field is highlighted with an orange box. Below the form, there are two text boxes: a pink one containing "情報基盤センターから送付された利用者ID (tUVXYZ)" and a yellow one containing "情報基盤センターから送付された初期パスワード". The browser's taskbar at the bottom shows various application icons and the system tray with the date "2020/04/15" and time "20:47".

Oakbridge-CX 利用支援ポータル

[ English/Japanese ]

ログイン

ログイン

ユーザー名とパスワードを入力して「ログイン」ボタンをクリックしてください。

ユーザー名:  ログイン

パスワード:  リセット

情報基盤センターから送付された利用者ID (tUVXYZ)

情報基盤センターから送付された初期パスワード

Copyright 2019 FUJITSU LIMITED

20:47  
2020/04/15

## ②初期パスワードの変更

Oakbridge-CX 利用支援ポータル

https://obcx-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal\_u.ja/index.cgi

Oakbridge-CX 利用支援ポータル

ログアウト

### パスワード変更

本機能で変更可能なパスワードは、Oakbridge-CXシステムの利用支援ポータル用パスワードです。

現在のパスワード

新しいパスワード

新しいパスワード(再入力)

変更

情報基盤センターから送付された初期パスワード

変更後のパスワードを入力 (2回)

### パスワード規約

- 8文字以上，現在と3文字以上異なる
- 2世代前までと異なる
- 英字(小文字，大文字)，数字，特殊文字各1字以上
- Linux辞書に登録されている語は不可
- 全角文字不可

Copyright 2019 FUJITSU LIMITED

18:38  
2020/04/14



## ③公開鍵登録 (id\_rsa.pub)

Oakbridge-CX 利用支援ポータル

https://obcx-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal\_uja/index.cgi

Oakbridge-CX 利用支援ポータル

ログアウト

お知らせ

SSH公開鍵登録

メール転送設定

パスワード変更

トークン表示

ディスク使用量表示

プリポスト予約

ドキュメント閲覧

OSS

SSH公開鍵登録

公開鍵を削除しました。

登録方式

直接入力

ファイルアップロード

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQDa6InmOYYaCrWjQDukjiNEfdW8veUwJyZtEI3oDu0
A28eey6p0wbtI7JB09xnI17O7HG4yYvOM81+/nIAHy5tAfJly0dsPzjTgdTBLdgi3cSf5pWEY6U9
6yaEr0Ei8Wge1HkXrhcwUjGDVTzvT0Refe6zLdRziL/KNmmesSQfR5lsZ/ihsjMgFxFxGaKsHHq
/IErCtHIII9V/Ds2yj6vkAaWH6asBn+ZsRiRFvWHPPhkYAnp/j3LY6b8Qfqg0p4WZRenh
/HgySWTYIGi8x67VzMaUIm9qIK0QFMCaK2rivX1fmbwyWJ
/vrWDqiek6YXoxLDu+GPeQ4CPvxJcZnqF9gf3
```

登録

1. 「SSH公開鍵登録」を選択
2. 先ほどCopyした公開鍵 ( id\_rsa.pub ) を貼り付ける
3. 「登録」をクリック

よくトラブルが起きますので、先に進めなければ質問してください



# スパコンには複数の公開鍵を登録できる

The screenshot shows a web browser window displaying the 'Oakbridge-CX 利用支援ポータル' (Oakbridge-CX User Support Portal). The page title is 'SSH公開鍵登録' (SSH Public Key Registration). A red arrow points to the 'SSH公開鍵登録' link in the left sidebar. The main content area shows a table of registered public keys, with two keys listed and highlighted by a red box. Below the table, there are radio buttons for '直接入力' (Direct Input) and 'ファイルアップロード' (File Upload). The footer of the page indicates 'Copyright 2019 FUJITSU LIMITED'.

登録されている公開鍵	ssh-rsa AAAAB3NzaC.....JcZnqF9gf3	表示	削除
	ssh-rsa AAAAB3NzaC.....pWGVie6w==	表示	削除

登録方式

直接入力

ファイルアップロード

# ポータルサイトでのマニュアル等閲覧 (1/2)

Oakbridge-CX 利用支援ポータル

https://obcx-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal\_u.ja/index.cgi

Oakbridge-CX 利用支援ポータル ログアウト

## ドキュメント閲覧の利用について

Oakbridge-CX マニュアルの Web 閲覧サービスを利用するにあたっては、以下の禁止事項を遵守していただきます。

- 核兵器又は生物化学兵器及びこれらを運搬するためのミサイル等の大量破壊兵器の開発、設計、製造、保管及び使用等の目的に利用しない。
- スーパーコンピュータの利用が認められた利用者本人のみが利用し、他者には利用させない。
- 本マニュアルの情報（印刷、コピーしたものを含む）を、利用者以外に開示または提供しない。
- 当センターが上記条項の違反、その他不正使用を検知した場合、当センターは利用者の Web 閲覧サービスの利用を直ちに停止することができる。また、利用者はこれに対して一切異議を唱えない。

上記禁止事項を  
遵守する

Copyright 2019 FUJITSU LIMITED

ページ内検索 すべて強調表示(A) 大文字/小文字を区別(C) 発音区別符号を区別(U) 単語単位(W)

# ポータルサイトでのマニュアル等閲覧 (2/2)

Oakbridge-CX 利用支援ポータル

https://obcx-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal\_uja/index.cgi

Oakbridge-CX 利用支援ポータル ログアウト

- お知らせ
- SSH公開鍵登録
- メール転送設定
- パスワード変更
- トークン表示
- ディスク使用量表示
- プリポスト予約
- ドキュメント閲覧**
- OSS

## Oakbridge-CX 利用手引書

ドキュメント名	言語	最新更新日
Oakbridge-CX システム利用手引書	日本語	2019/10/01
Oakbridge-CX グループコース プロジェクト管理者用利用手引書	日本語	2019/07/03

## 製品マニュアル

### インテルParallel Studio XE 2019

ドキュメント名	言語	最新更新日
スタートアップガイド	日本語 英語	2019/07/01
Fortranコンパイラ19.0 スタートアップガイド	日本語 英語	2019/07/01
C++コンパイラ19.0 スタートアップガイド	日本語 英語	2019/07/01

### インテルMPIライブラリ 2019

ドキュメント名	言語	最新更新日
スタートアップガイド	英語	2019/07/01

### インテルMKL 2019

ドキュメント名	言語	最新更新日
スタートアップガイド	日本語 英語	2019/07/01

Copyright 2019 FUJITSU LIMITED

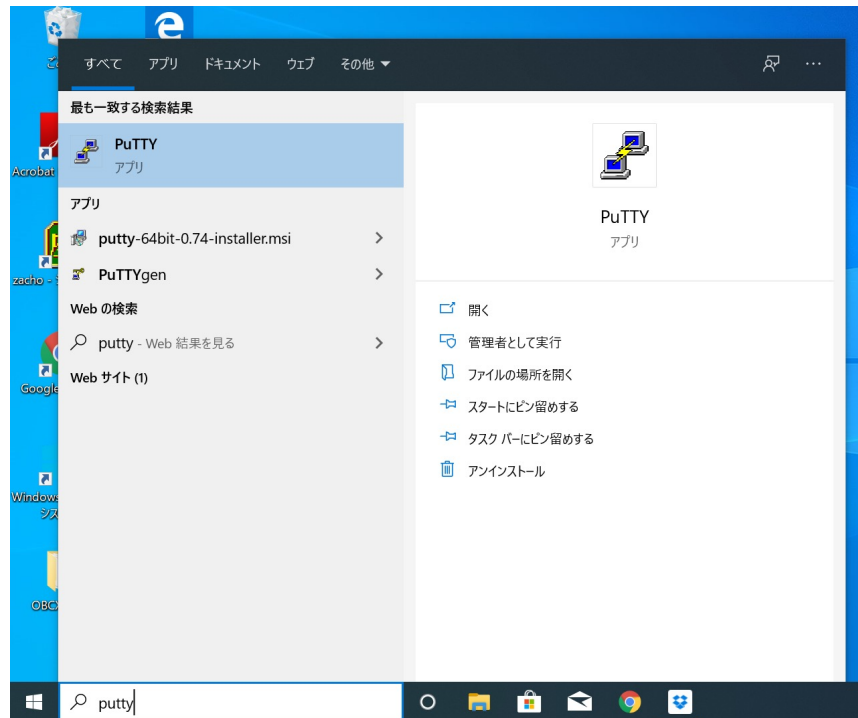
ページ内検索  すべて強調表示(A) 大文字/小文字を区別(C) 発音区別符号を区別(I) 単語単位(W)

22:37 2020/04/15

# スパコンにログイン

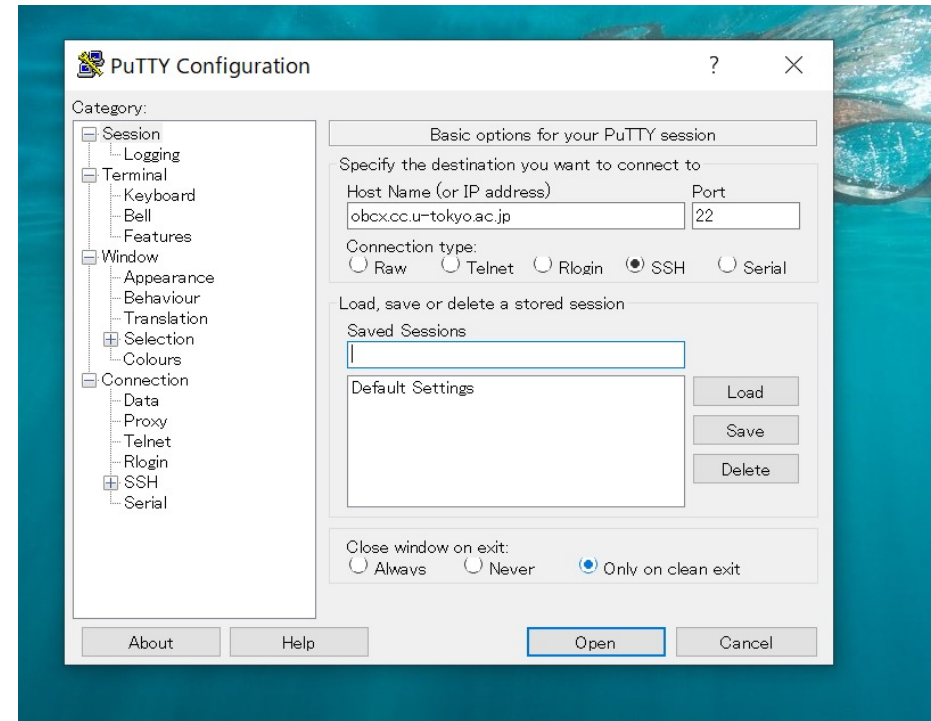
## (1) PuTTYを起動

アイコンがなければ左下のメニューから検索



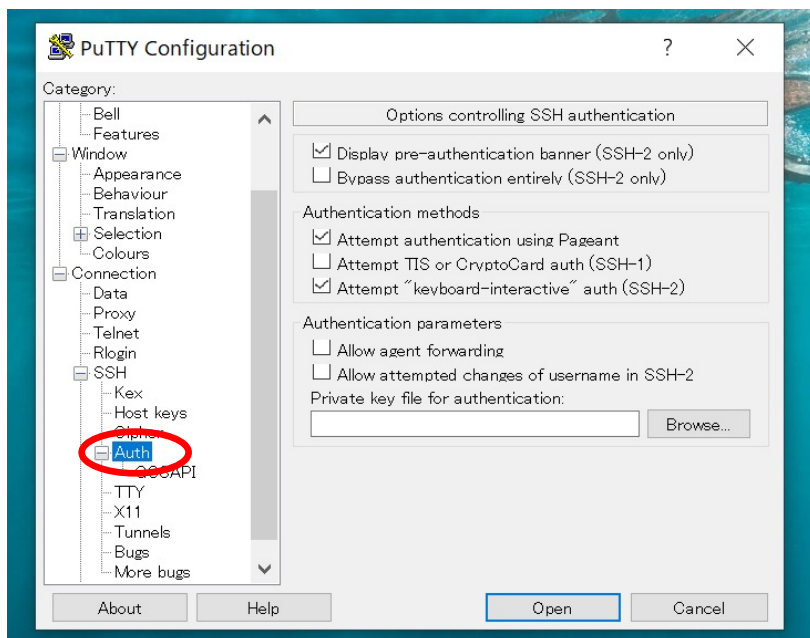
## (2) 接続先 (Host Name) を設定

[obcx.cc.u-tokyo.ac.jp](http://obcx.cc.u-tokyo.ac.jp)

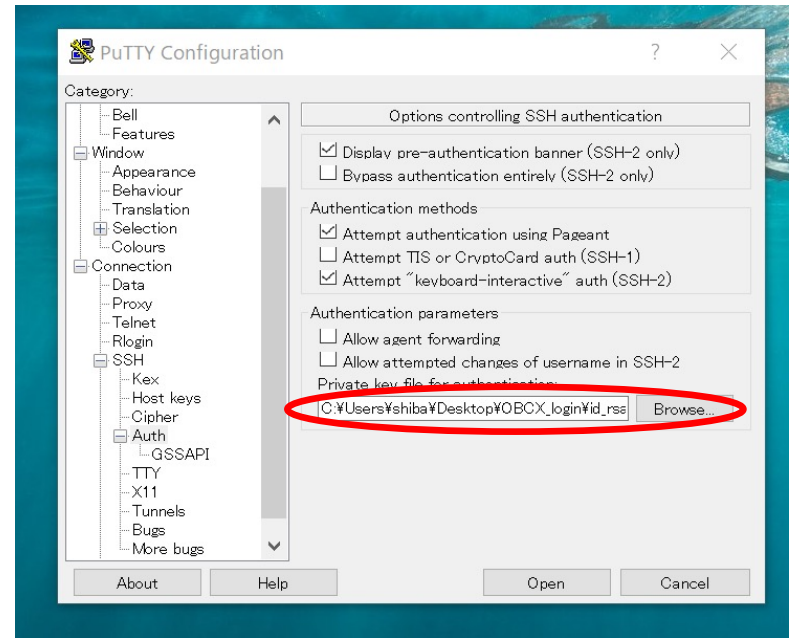


# スパコンにログイン

## (3) 秘密鍵を指定する



Connection → SSH → Auth  
と移動



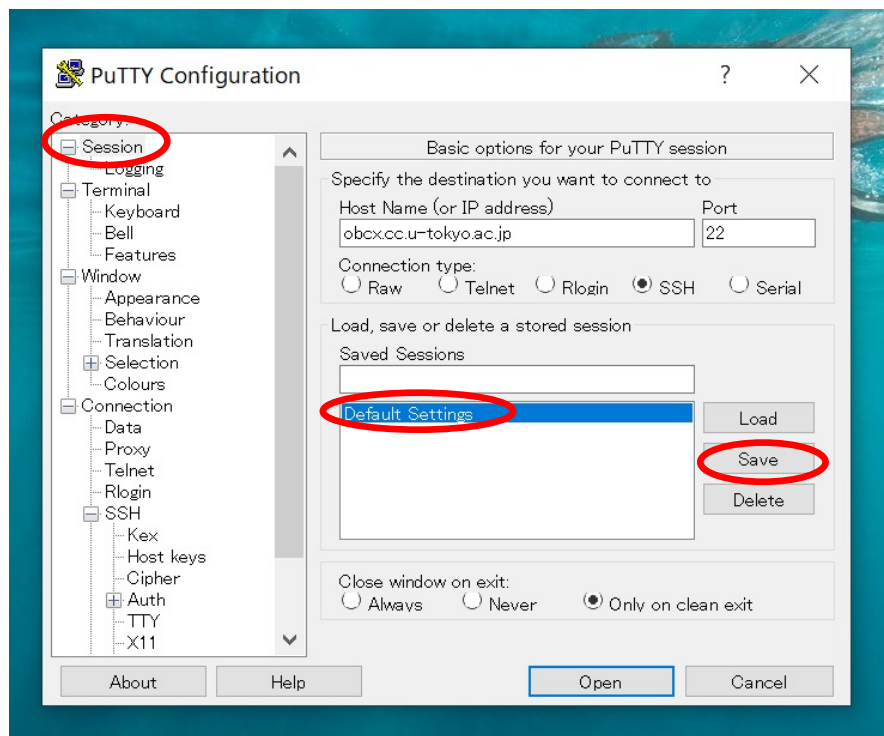
先ほど保存した秘密鍵を指定  
“id\_rsa”



# スパコンにログイン

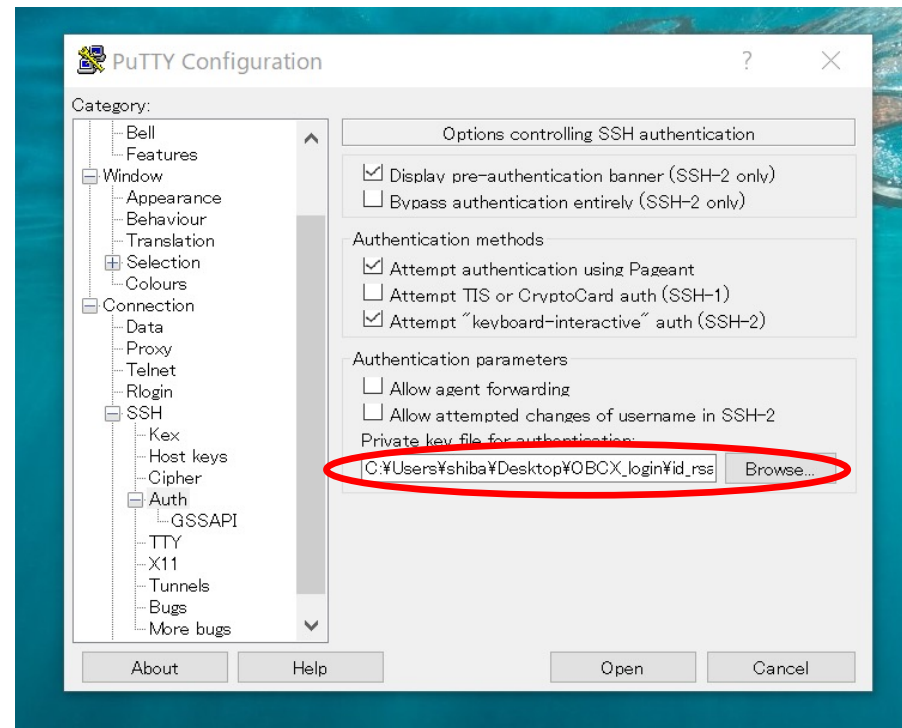
## (4) 設定を保存

左上“Session”へ移動  
→ “Default Setting”を選択, Save



## (5) 実際にログイン

先ほど作成保存した秘密鍵を指定  
“id\_rsa”



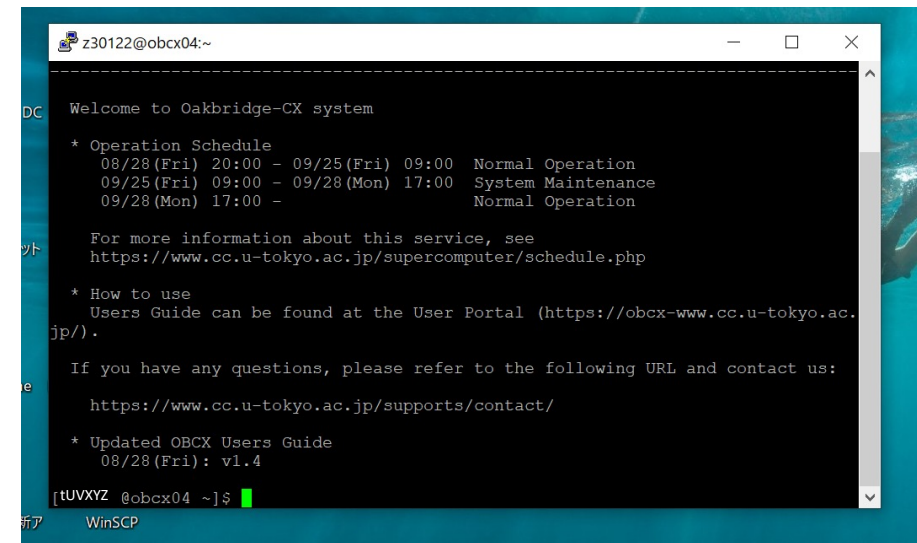
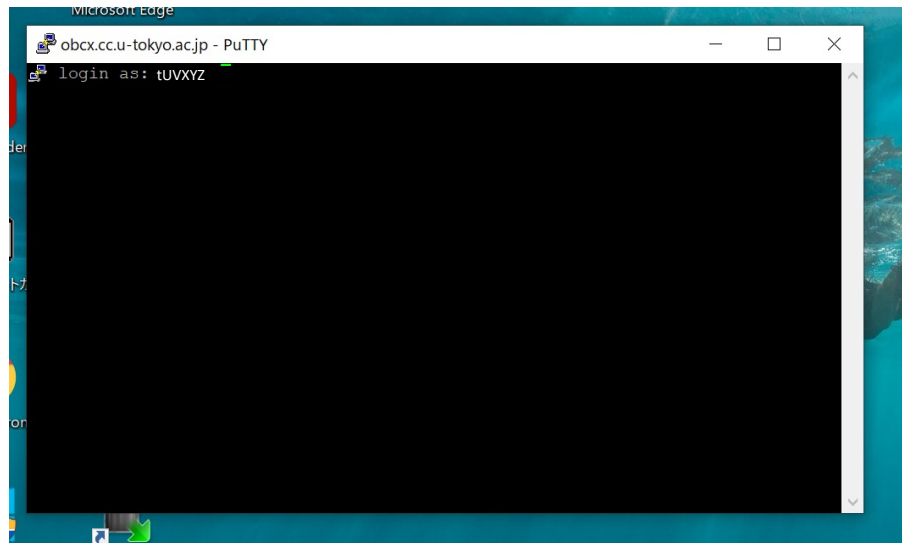
次回以降すぐLoadで開始できます

# スパコンにログイン

## (5) 実際にログイン〔続〕


ユーザー名 ( **tUVXYZ** ) を入力

スパコンのログインノード上で  
操作ができるようになりました。



# ログインしたら

ログインノード上のLinux の操作画面になります  
→ 命令文を打ちます

```
[tUVXYZ@obcx01 ~]$ pwd   
/home/tUVXYZ
```

pwd — 現在のディレクトリ(フォルダ)  
を位置を返すコマンド



# 休憩

余裕のあるかたは次のページ以降を  
少し予習してください

# スーパーコンピューター (スパコン) 利用のイメージ

通常、スパコンでは  
ログインノードと呼ば  
れる玄関口から実行  
命令を出します



端末

1. ログイン  
SSH

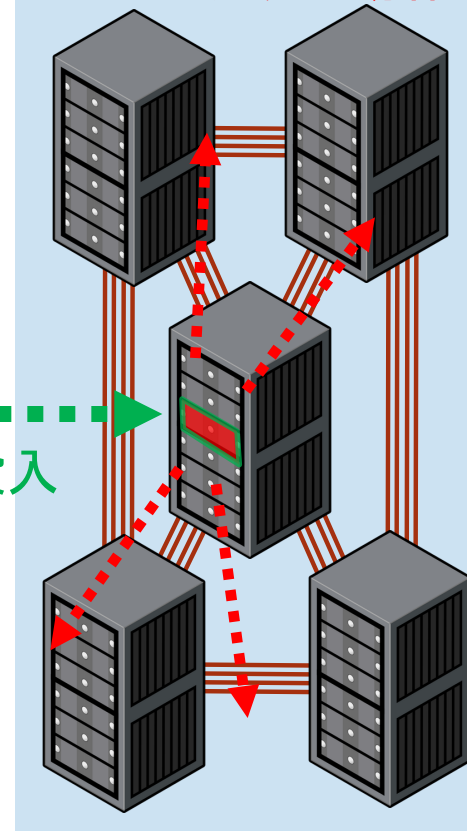


ログインノード

2. ジョブ投入

```
[tut138@obcx02~]$ ls -l  
drwxr-x--- 2 shiba group 10 1  
Apr 13:00 test.out  
[tut138@obcx02~]$ ./test.out  
Hello world  
[tut138@obcx02~]$ qsub a.sh
```

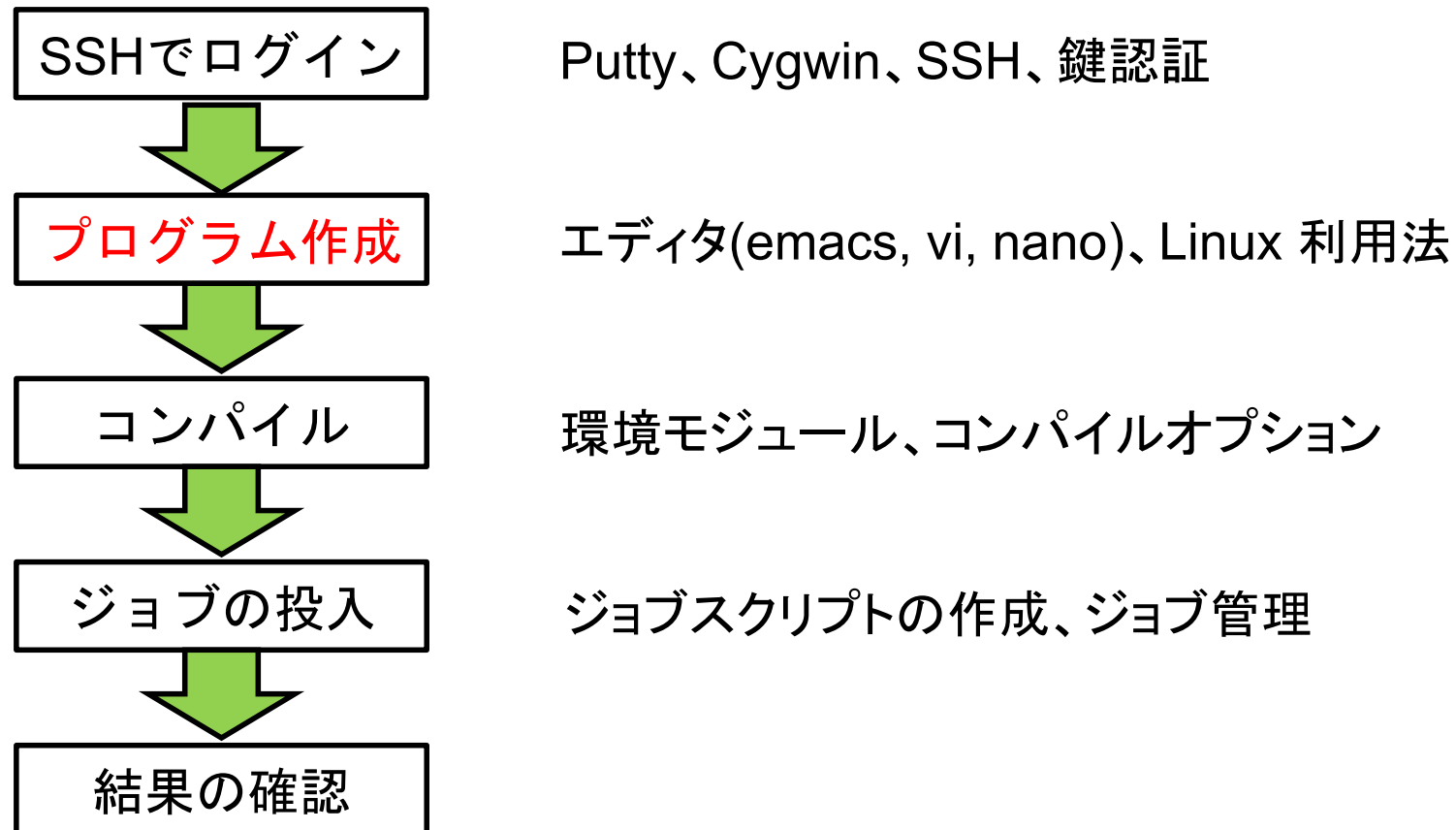
3. プログラム動作



計算ノード  
= 本体

SSH で接続するには  
鍵の準備が必要！

# スパコンを使うための手順



# コマンドとは？

## 特定の機能を実行する命令

\$コマンド [オプション1、オプション2...] 引数1、引数2.....

例     \$ cd /home/t00270/           (作業ディレクトリを変更)  
       \$ cp a.txt b.txt  
          (a.txtと同じ内容のファイルを b.txt として作成)

### コマンドの例

コマンド	用途	よく使うオプション
pwd	現在のディレクトリを表示	
cd	ディレクトリを変更	
ls	ファイル一覧を表示	-l : 詳細表示
cp	ファイルまたはディレクトリのコピー	-r : 再帰的にコピー
rm	ファイルまたはディレクトリの削除	-r : 再帰的に削除
exit	セッションの終了 (スパコンからログアウト)	

## ほかのコマンドの例（簡単なもの）

コマンド	用途	よく使うオプション
mv	ファイルまたはディレクトリの移動	
rm	ファイルまたはディレクトリの削除	-r : 再帰的に削除
mkdir	ディレクトリの作成	
man	コマンドの説明	
cat	ファイル内容の表示（全体）	
less	ファイル内容の表示（一画面ごと）	
head	ファイル内容の表示（先頭部分）	-n ### : 表示行数の指定
tail	ファイル内容の表示（末尾部分）	-n ### : 表示行数の指定
grep	ファイルに含まれる文字列を検索	
diff	2つのファイルの内容比較	
echo	変数や文字列の値を出力	
exit	セッションの終了	

# Bash特有の機能

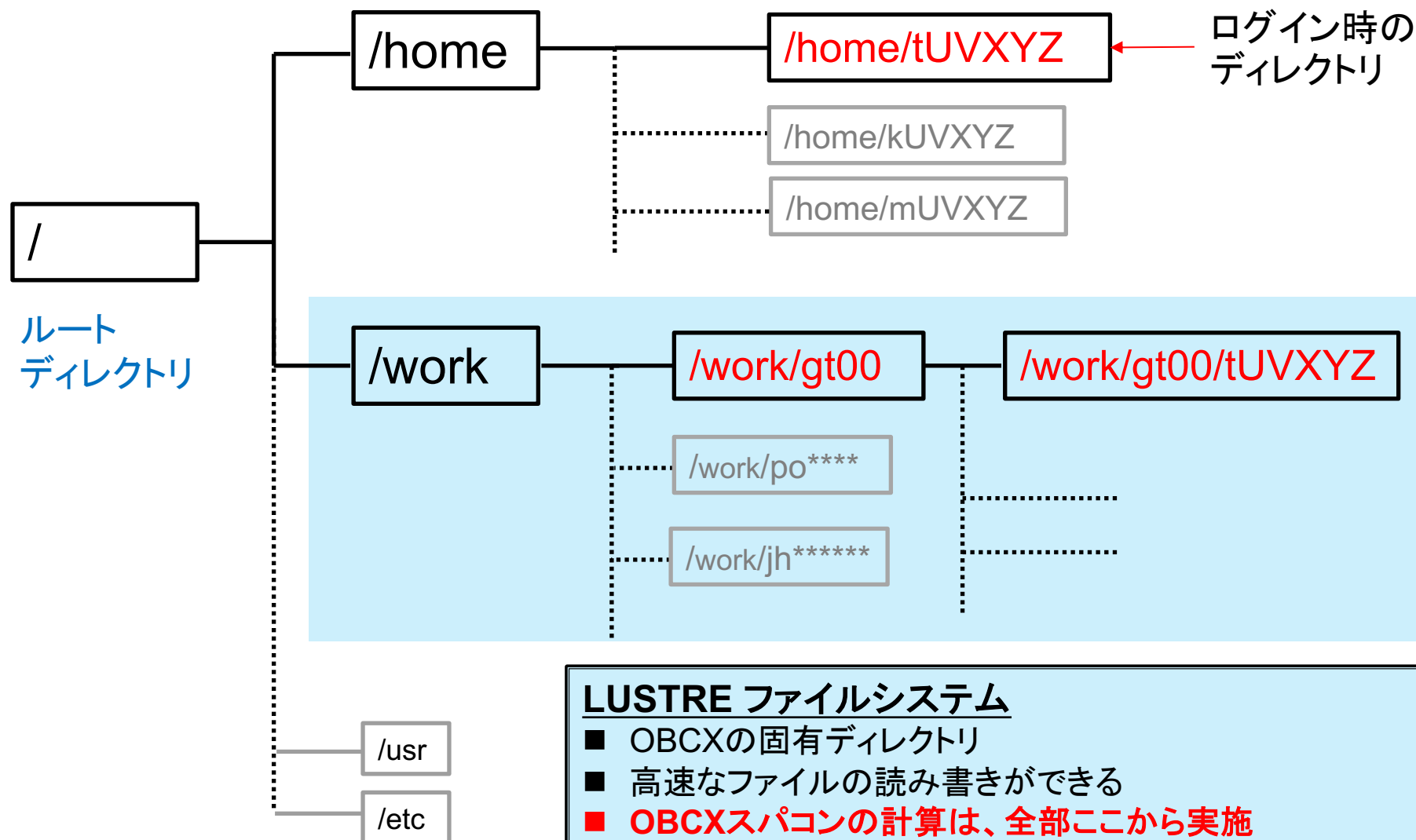
OBCXシステムのデフォルトシェルはbashです。

## コマンド入力を簡易にする基本機能一覧

	操作	動作内容
Auto fill	tab	途中まで入力したコマンドやパスの残りを補完
履歴	↑	過去に実行したコマンドを1つずつさかのぼる
履歴探索モード	ctrl+r	過去に実行したコマンドを検索
リダイレクト	>	結果をファイルなどに出力 例：ls > list.txt list.txtにlsの結果が出力される
パイプ		結果を次のコマンドに渡す 例：ls   sort lsコマンドの結果をソートする

コマンドとbashの機能を使用したスクリプトも記述可能

# Linux(OBCX)でのディレクトリ構造



## LUSTRE ファイルシステム

- OBCXの固有ディレクトリ
- 高速なファイルの読み書きができる
- **OBCXスパコンの計算は、全部ここから実施**
  - **Home上での実行は不可**
- トラブルもたまにあるので /homeから隔離されています

# コマンドの実行例

## コマンドの実行例

```
[ tUVXYZ @obcx05 ~]$ pwd 現在のディレクトリを出力
/home/ tUVXYZ
[ tUVXYZ @obcx05 ~]$ cd /work/gt00/ tUVXYZ 講習会用ディレクトリに移動
[ tUVXYZ @obcx05 tUVXYZ ]$ cd ../ 1つ上のディレクトリに移動
[ tUVXYZ @obcx05 gt00]$ pwd
/work/gt00
[ tUVXYZ @obcx05 gt00]$ cd ~/ ホームディレクトリに移動
[ tUVXYZ @obcx05 ~]$ pwd
/home/ tUVXYZ
[ tUVXYZ @obcx05 ~]$ cd /work/gt00/ tUVXYZ
[ tUVXYZ @obcx05 tUVXYZ ]$ mkdir test testディレクトリを作成
[ tUVXYZ @obcx05 tUVXYZ ]$ ls 現在のディレクトリにあるファイルおよび
test ディレクトリ一覧を表示
[ tUVXYZ @obcx05 tUVXYZ ]$
```



# Linux OS

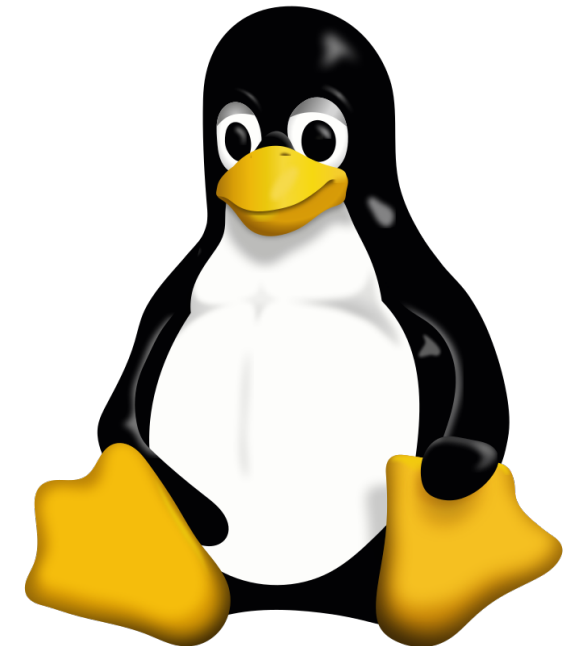
## 世界中のスパコンのほとんどが採用しているOS

- 現在のTOP500(上位500台のスパコンランキング)のシェアは100%
- スパコンだけでなく、サーバーの多くはLinuxで動作
- 基本的に**OpenSource**で構成
- CLI(Command Line Interface)で完結、高機能なShellが用意されている。
  - 本講習ではBashを使用
  - GUI(Graphical User Interface)もあり。  
スパコンではあまり使わない。

講習会では基本的なコマンドとBashについて扱います。

Linuxではフォルダをディレクトリといいます。  
(この後頻出します。)

Tux



# エディタ

## ファイルを編集するためのソフトウェア

Windowsの標準だとメモ帳に該当

Linuxでは以下のエディタが有名（ほかにもたくさんあります）

### ■ Emacs

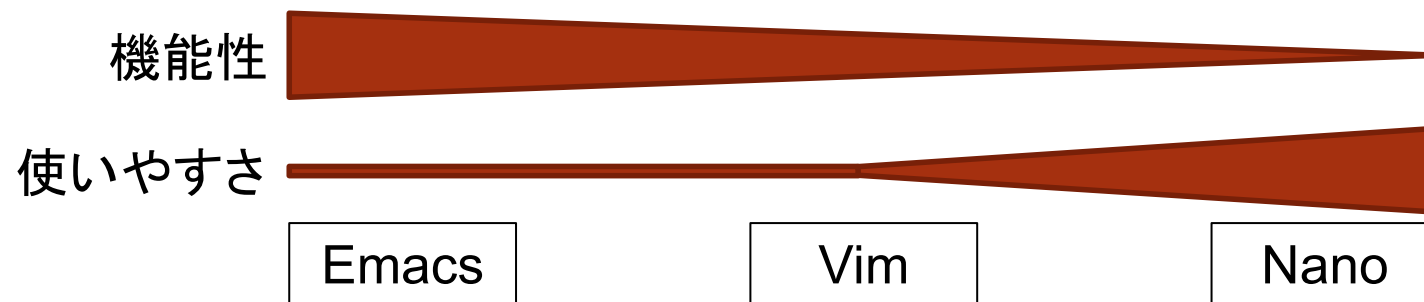
- 高機能
- 拡張機能も豊富 (LISPで記述されている)

### ■ Vim

- 軽量
- Linuxの初期状態でインストールされている

### ■ Nano

- 一番簡易
- コマンドが下段に表示されるため、わかりやすい



# ログインしたら

```
$ pwd 
```

```
/home/tUVXYZ
```

```
$ cd /work/gt00/tUVXYZ 
```

```
$ pwd 
```

```
/work/gt00/tUVXYZ
```

```
$ cd 
```

```
$ pwd 
```

```
/home/t00XYZ
```

1. ログインしたら「/home/tUVXYZ」に入る
2. /homeは容量が少ないので「/work/gt00/tUVXYZ」に移動すること
3. 「cd」でホームに戻れます

# プログラム例題：フィボナッチ数列

漸化式で定義

$$F_0 = 1$$

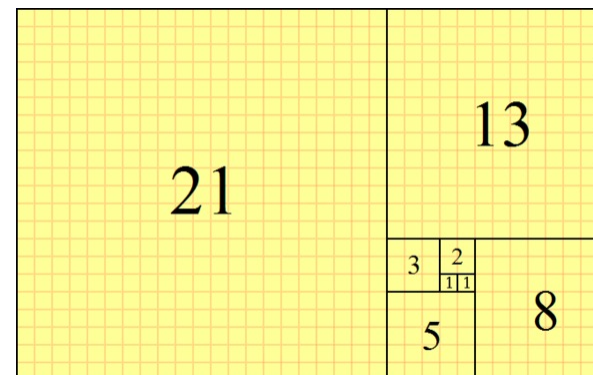
$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$

繰り返し計算で数列が得られる

1, 1, 2, 3, 5, 8, 13, 21 ....

繰り返し計算はスーパーコン  
ピューター利用の出発点



From Wikipedia

<https://ja.wikipedia.org/wiki/フィボナッチ数>

# プログラム例題：フィボナッチ数列 92個

1	75025	7778742049	806515533049393
1	121393	12586269025	1304969544928657
2	196418	20365011074	2111485077978050
3	317811	32951280099	3416454622906707
5	514229	53316291173	5527939700884757
8	832040	86267571272	8944394323791464
13	1346269	139583862445	14472334024676221
21	2178309	225851433717	23416728348467685
34	3524578	365435296162	37889062373143906
55	5702887	591286729879	61305790721611591
89	9227465	956722026041	99194853094755497
144	14930352	1548008755920	160500643816367088
233	24157817	2504730781961	259695496911122585
377	39088169	4052739537881	420196140727489673
610	63245986	6557470319842	679891637638612258
987	102334155	10610209857723	1100087778366101931
1597	165580141	17167680177565	1779979416004714189
2584	267914296	27777890035288	2880067194370816120
4181	433494437	44945570212853	4660046610375530309
6765	701408733	72723460248141	7540113804746346429
10946	1134903170	117669030460994	
17711	1836311903	190392490709135	
28657	2971215073	308061521170129	
46368	4807526976	498454011879264	

パソコンでも計算は一瞬

# エディタでプログラムを作成する

さて、実際に作業してみましよう。  
プログラムを編集するディレクトリを作成します。

```
[tUVXYZ@obcx01 ~]$ cd   
[tUVXYZ@obcx01 ~]$ mkdir fibonacci   
[tUVXYZ@obcx01 ~]$ cd fibonacci 
```

ログインノード上では、Emacs, vim, または nano のエディタを利用してください。

例1: Emacs を使用, Fortranプログラムを作成

```
[tUVXYZ@obcx01 fibonacci]$ emacs fibonacci.f90 
```

例2: vim を使用, Python プログラムを作成

```
[tUVXYZ@obcx01 fibonacci]$ vim fibonacci.py 
```

例3: nano を使用, C プログラムを作成

```
[tUVXYZ@obcx01 fibonacci]$ nano fibonacci.c 
```

# Emacs チートシート

## \$ emacsで起動

Emacs = 「control (ctl) キー + 文字」でコマンドに移行

コマンド	
Ctl-x Ctl-s	編集中のファイルを同じファイル名で保存
Ctl-x Ctl-w	編集中のファイルを名前をつけて保存
Ctl-x Ctl-f	既存のファイルを開く
Ctl-x Ctl-c	Emacs を終了する (保存するか聞かれたらY/N で答える)
Ctl-g	コマンド入力
Ctl-a	行の先頭にカーソルを持ってくる
Ctl-k	カーソルから行末までを削除し、 (クリップボードみたいなところに) コピー
Ctl-y	コピーされた内容を貼り付ける
C-space	Mark Set
Mark Set, 範囲選択, M, w	コピー
Mark Set, 範囲選択, C-w	切り取り
Ctl-s	文字列を検索する
Esc-%	文字列を置換する



コマンド確認画面

# vim チートシート

## \$ vi で起動

ノーマルモード / 入力モード / コマンドを切り替え

現在のモード	移行先モード	キー
ノーマル	入力	i, a, Ins
ノーマル	コマンドライン	:
入力	ノーマル	esc, ctrl-c

キー	
Esc	ノーマルモードに移行する
i	入力モード（挿入）に移行する
o	新しい行を追加して入力モード（挿入）に移行
R	入力モード（上書き）に移行する
:w ###	コマンド - ファイル名 ### で保存する
:q	コマンド - vim を終了する
:q!	コマンド - 保存せずにvim を強制終了する
/###	### という文字列を検索する
yy	カーソルのある行をコピーする
p	貼り付けする

### 編集画面

```

VIM - Vi Improved
      version 8.2.905
      by Bram Moolenaar 他.
      Modified by <bugzilla@redhat.com>
      Vim はオープンソースであり自由に配布可能です

      Vimの開発を応援してください!
      詳細な情報は      :help sponsor<Enter>

      終了するには      :q<Enter>
      オンラインヘルプは :help<Enter> か <F1>
      バージョン情報は  :help version8<Enter>
  
```

モードおよびコマンド確認画面

- ブランク → ノーマルモード
- : → コマンドモード
  - :w Ret 上書き保存
- --挿入-- → 入力モード



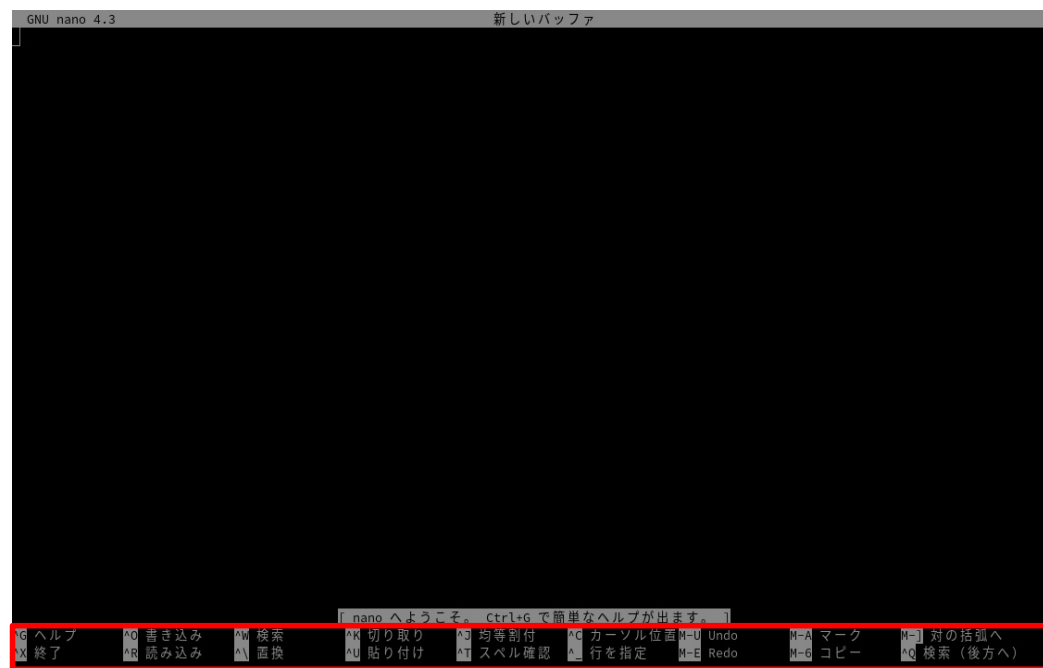
# nano チートシート

nano = 最も操作方法が単純      一予習漏れの場合はお使いください。

## \$ nano で起動

- M-U → Escキー、Uキーを順番に押す

キー	
Ctrl+S	ファイルを保存
Ctrl+O	ファイルを名前をつけて保存
Ctrl+X	nano エディタ を終了する
Ctrl+W	文字列を検索 (順方向)
Ctrl+D	カーソルのある文字を削除
M-A	マーク
M-6	コピー
Ctrl+K	1行カットして削除
Ctrl+U	ペースト



コマンド確認画面

- ^ → Ctl
- M → Esc
- - → 順次押す

# プログラム例題：フィボナッチ数列

## C言語

プログラム作成

[tUVXYZ@obcx01 fibonacci]\$ **emacs fibonacci.c**

Return

```
#include <stdio.h>

int main(void) {
    int i;
    long a, b, tmp;

    a=1;
    b=1;
    printf("%ld¥n", a);

    for (i=2; i<=92; i++) {
        tmp = b;
        b = a + b;
        a = tmp;
        printf("%ld¥n", a);
    }
}
```

注) ¥ マークがあればバックスラッシュと同じ  
と思って(今は)結構です。

# プログラム例題：フィボナッチ数列

## Fortran言語

プログラム作成

[tUVXYZ@obcx01 fibonacci]\$ **emacs fibonacci.f90**

Return

```
program main

  implicit none
  integer(kind=4) i
  integer(kind=8) a,b,tmp

  a = 1
  b = 1

  do i=2, 92
    tmp = b
    b= a + b
    a = tmp
    print *, a
  end do

end program main
```

# プログラム例題：フィボナッチ数列

## Python

プログラム作成

```
[tUVXYZ@obcx01 fibonacci]$ emacs fibonacci.py
```

Return

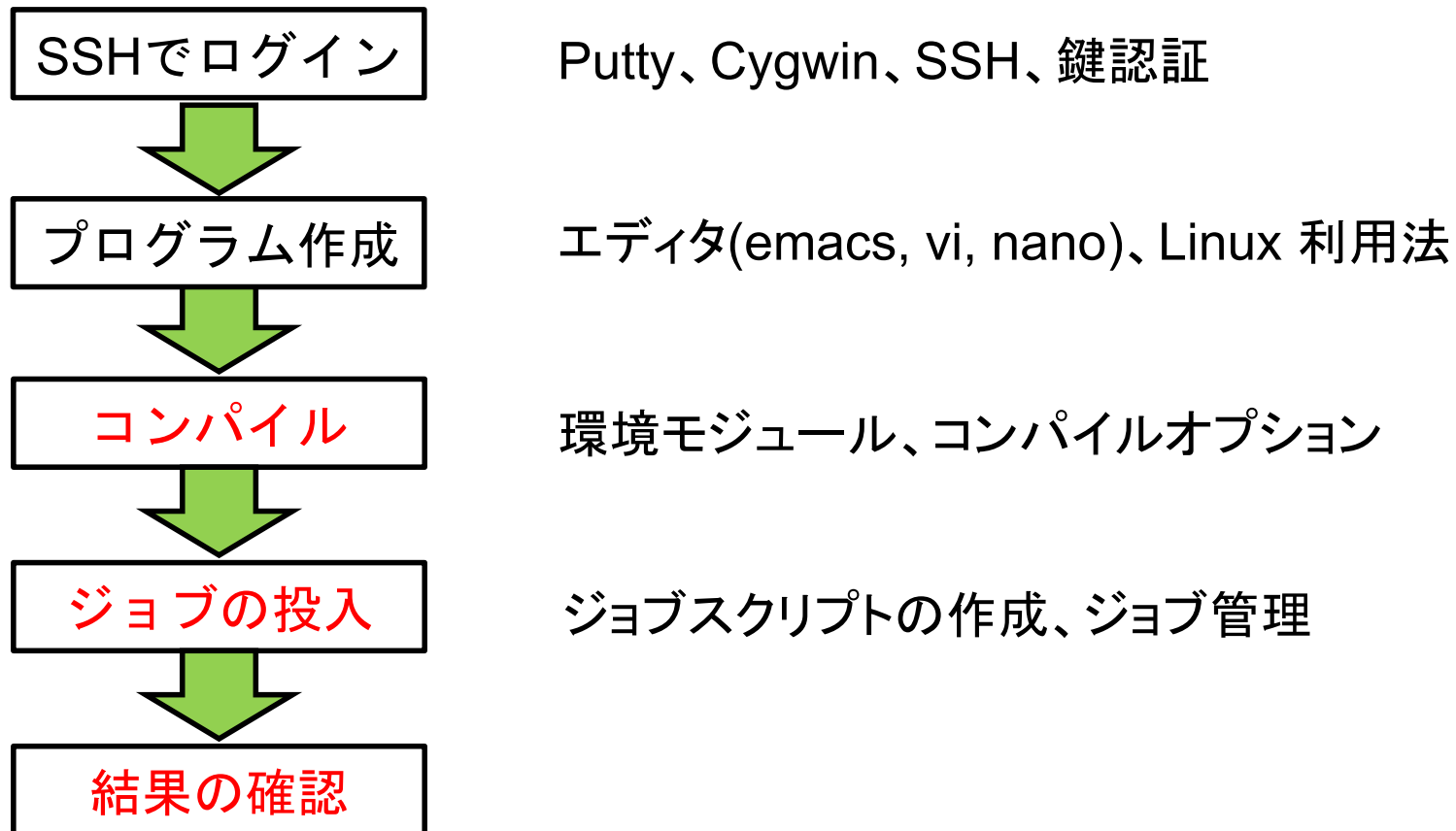
```
a, b = 1, 1
print(a)

for i in range(1,92):
    a, b = b, a+b
    print(a)
```

注) python では

- 64ビット整数など変数型の選択を自動的にやってくれます
- 多変数を同時に代入計算できます。
- **コンパイルはする必要ありません。**

# スパコンを使うための手順



# Environmental modulesの利用 (1)

## コンパイラ等の選択

gnu コンパイラ, インテル製コンパイラ, python インタープリター, ...

## ライブラリを利用した計算の高速化

離散フーリエ変換, 線形代数・連立1次方程式, ...

## 使用中モジュールの表示

```
[tUVXYZ@obcx01 ~]$ module list   
Currently Loaded Modulefiles:  
1) impi/2019.9.304      2) intel/2020.4.304
```

Intel MPI ライブラリ

Intel コンパイラ

## コンパイラ, MPI ライブラリを切り替えるために一度 消去します

```
[tUVXYZ@obcx01 ~]$ module purge   
[tUVXYZ@obcx01 ~]$ module list   
No Modulefiles Currently Loaded.
```

# 利用可能なenvironmental modulesの一覧を表示できます

```
[tUVXYZ@obcx01 ~]$ module avail Return
```

```
----- /home/opt/local/modulefiles/L/mpi/intel/2019.5.281/mpi/2019.5.281 -----
alps/2.3.0(default)          phdf5/1.10.5(default)
feram/0.26.04(default)       pnetcdf/1.11.2(default)
frontflow_blue/8.1(default)  ppohBEM/0.5.0(default)
frontistr/4.5(default)       ppohDEM_util/1.0.0(default)
modylas/1.0.4(default)       ppohFDM/0.3.1(default)
mpi-fftw/3.3.8(default)      ppohFEM/1.0.1(default)
netcdf-fortran-parallel/4.4.5(default) ppohFVM/0.3.0(default)
netcdf-parallel/4.7.0(default) pt-scotch/6.0.6(default)
openmx/3.8(default)          revocap_coupler/2.1(default)
parmetis/4.0.3(default)      superlu_dist/6.1.1(default)
petsc/3.11.2(default)        xtapp/rc-150401(default)
phase/2019.01(default)
```

```
----- /home/opt/local/modulefiles/L/compiler/intel/2019.5.281 -----
R/3.6.0(default)            metis/5.1.0(default)
akaikkr/cpa2002v010(default) mt-metis/0.6.0(default)
bioconductor/3.10(default)  netcdf/4.7.0(default)
blast/2.9.0(default)        netcdf-cxx/4.3.0(default)
bwa/0.7.17(default)         netcdf-fortran/4.4.5(default)
fftw/3.3.8(default)         paraview/5.6.1(default)
gsl/2.5(default)            povray/3.7.0.8(default)
hdf5/1.10.5(default)        ppohAT/1.0.0(default)
hdf5/1.8.21                 revocap_refiner/1.1.04(default)
impi/2019.5.281(default)    samtools/1.9(default)
intelpython/2.7             scotch/6.0.7(default)
intelpython/3.6(default)    superlu/5.2.1(default)
mesa/19.0.6(default)        superlu_mt/3.1(default)
metis/4.0.3                 xabclib/1.03(default)
```

```
----- /home/opt/local/modulefiles/L/core -----
acusolve/2019.1.0(default)  intel/2018.3.222
advisor/2019.3.0.591490     intel/2019.3.199
advisor/2019.4.0.597843     intel/2019.4.243
advisor/2019.5.0.602216(default) intel/2019.5.281(default)
anaconda/2-2019.03         intel/2020.1.217
anaconda/3-2019.03(default) itac/2019.4.036
bioperl/1.007002(default)  itac/2019.5.041(default)
bioruby/1.5.2(default)     julia/1.4.0(default)
cmake/3.0.2                 llvm/7.1.0(default)
cmake/3.14.5(default)       massivethreads/0.97
```



# OBCX ではmodule 依存関係が複雑 → 簡易表示コマンドがあります

```
[tUVXYZ@obcx01 ~]$ show_module
```

ApplicationName	ModuleName	Node	BaseCompiler/MPI
ALPS	alps/2.3.0	login, compute	intel/2020.4.304/impi/2019.9.304
Acusolve	acusolve/2019.1.0	login, compute	-
Advisor	advisor/2019.4.0.597843	login, compute	-
Advisor	advisor/2019.5.0.602216	login, compute	-
Advisor	advisor/2020.3.0.607294	login, compute	-
Advisor	advisor/2019.3.0.591490	login, compute	-
AkaiKKR	akaikkr/cpa2002v010	login, compute	intel/2020.4.304
Anaconda	anaconda/2-2019.03	login, compute	-
Anaconda	anaconda/3-2019.03	login, compute	-
Arm DDT	ddt/19.1	compute	-
Arm DDT	ddt/20.2.1	compute	-
Arm DDT	ddt/20.0.2	compute	-
BLAST	blast/2.11.0	login, compute	intel/2020.4.304
BWA	bwa/0.7.17	login, compute	intel/2020.4.304
BioPerl	bioperl/1.007002	login, compute	-
BioRuby	biорuby/1.5.2	login, compute	-
Bioconductor	bioconductor/3.10	login, compute	intel/2020.4.304
CMake	cmake/3.0.2	login, compute	-
CMake	cmake/3.14.5	login, compute	-
CP2K	cp2k/v8.1	login, compute	intel/2020.4.304/impi/2019.9.304
Devtoolset	devtoolset/7	login, compute	-
FFTW	fftw/3.3.8	login, compute	intel/2020.4.304
FFTW	mpi-fftw/3.3.8	login, compute	intel/2020.4.304/impi/2019.9.304
FeRAM	feram/0.26.04	login, compute	intel/2020.4.304/impi/2019.9.304
GATK	gatk/4.1.2.0	login, compute	-
GCC	gcc/4.8.5	login, compute	-
GCC	gcc/7.5.0	login, compute	-
GNU Octave	octave/6.1.0	login, compute	-
GNU Scientific Library	gsl/2.6	login, compute	intel/2020.4.304
GROMACS	gromacs/2020.5	login, compute	intel/2020.4.304/impi/2019.9.304
Go	go/1.12.6	login, compute	-
HDF5	hdf5/1.12.0	login, compute	intel/2020.4.304
HDF5	hdf5/1.8.22	login, compute	intel/2020.4.304
HDF5	phdf5/1.12.0	login, compute	intel/2020.4.304/impi/2019.9.304
HDF5	phdf5/1.8.22	login, compute	intel/2020.4.304/impi/2019.9.304
HyperWorks	hyperworks/2019.1.0	login, compute	-
IASP91	iasp91/default	compute	-
Inspector	inspector/2019.5.0.602103	login, compute	-
Inspector	inspector/2019.3.0.591484	login, compute	-

コンパイラと連動



対応するベースコンパイラの情報も一緒に表示されます



# コンパイル、実行環境構築

コンパイル、実行環境を整えるために、moduleコマンドを使用

\$ module [オプション] 引数

オプション	内容
avail	利用可能な環境の一覧を表示
list	現在ロードしている環境一覧を表示
load	指定した環境のロード
unload	指定した環境のアンロード
switch	環境のロードとアンロードを同時に実行
purge	環境を全てアンロード

例えば、Pythonを使用する場合、  
\$ module load python/3.7.3

## Environmental modulesの利用 (2)

Intel コンパイラの前のバージョンを load してみます

```
[tUVXYZ@obcx01 ~]$ module load intel/2020.1.217 Return
[tUVXYZ@obcx04 ~]$ module list Return
Currently Loaded Modulefiles:
 1) impi/2019.7.217    2) intel/2020.1.217
```

Intel MPI ライブラリも、一緒に自動でload されました

Python は比較的最近のバージョン3.7.3 を使えるようにしましょう。

```
[tUVXYZ@obcx01 ~]$ module load python/3.7.3 Return
[tUVXYZ@obcx01 ~]$ module list Return
Currently Loaded Modulefiles:
 1) impi/2019.7.217    2) intel/2020.1.217    3) python/3.7.3
```

# スーパーコンピュータ (スパコン) 利用のイメージ

通常、スパコンでは  
ログインノードと呼ば  
れる玄関口から実行  
命令を出します



端末

1. ログイン  
SSH

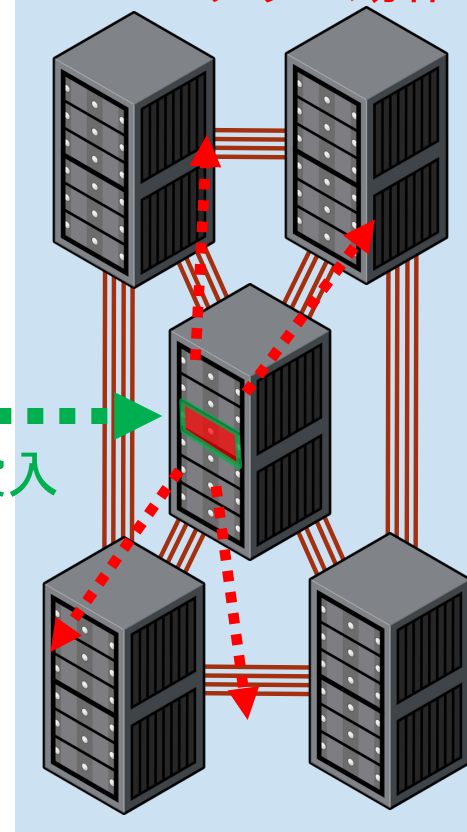


ログインノード

2. ジョブ投入

```
[tut138@obcx02~]$ ls -l  
drwxr-x--- 2 shiba group 10 1  
Apr 13:00 test.out  
[tut138@obcx02~]$ ./test.out  
Hello world  
[tut138@obcx02~]$ qsub a.sh
```

3. プログラム動作



計算ノード  
= 本体

SSH で接続するには  
鍵の準備が必要！

# ジョブスクリプト = スパコンへの指示書

プログラムをコンパイルしたあとにログインノード上で  
直接プログラム実行することはしないでください！

— 負荷がかかると、他のユーザーの同時作業の妨げとなります

かわりに計算ノードに計算をしてもらうための「指示書」が必要となります。

C, Fortran 向け (fibonacci.sh)

```
#!/bin/bash
#PJM -L rscgrp=tutorial [リソースグループ]
#PJM -L node=1 [使用するノードの数]
#PJM -L elapse=0:01:00 [実行時間上限(1分)]
#PJM -g gt00 [バジェットグループ名]
#PJM -N fibonacci [今回のジョブの名前]
#PJM -o stdout.txt [標準出力先ファイル名]
#PJM -j [エラー出力を標準出力にマージ]

module purge
module load intel/2020.1.217
./fibonacci.out
```

Python 向け (fibonacci.sh)

```
#!/bin/bash
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM -L elapse=0:01:00
#PJM -g gt00
#PJM -N fibonacci
#PJM -o stdout.txt
#PJM -j

module load python/3.7.3
python ./fibonacci.py
```

# ジョブスクリプトを用意しました。

ご自身で入力いただくと時間を要しますので、  
用意されたものをコピーしてください。

```
[tUVXYZ@obcx01 ~]$ cd  
[tUVXYZ@obcx01 ~]$ cd fibonacci
```

C, Fortran を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_c/fibonacci.sh .
```

Return

Python を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_python/fibonacci.sh .
```

Return

# エディターでの編集が間に合わなかった方は

サンプルとなるプログラムも用意してありますので、ご自身でコピーしてください。

C を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_c/fibonacci.c .
```

Return

Fortran を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_fortran/fibonacci.f90 .
```

Return

Python を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_python/fibonacci.py .
```

Return

# プログラムのコンパイル

C または Fortran を使用される方

→ プログラムをコンパイルして、実行ファイル(バイナリ)を生成します。

C を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ icc fibonacci.c -o fibonacci.out Return
```

プログラム名

実行ファイル名

Fortran を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ ifort fibonacci.f90 -o fibonacci.out
```

Return



# プログラムのコンパイル

## IntelおよびGNUコンパイラがインストール済み

ソースコードのタイプ	コンパイラ開発元	言語	呼び出しコマンド
非MPIコード	Intel	C	icc
		Fortran	ifort
	GNU	C	gcc
		Fortran	gfortran
MPIコード	Intel	C	mpiicc
		Fortran	mpiifort
	GNU	C	mpicc
		Fortran	mpif77 or mpif90

IntelでC言語のコードをコンパイルする場合

\$ icc [オプション] “ソースコード.c” -o “出力ファイル名”

GNUでFortran90/95のコードをコンパイルする場合

\$ gfortran [オプション] -free-line-length-none “ソースコード.f90” -o “出力ファイル名”

-free-line-length-none : Fortranの自由形式を有効化

# コンパイルオプション例（参考情報）

## IntelおよびGNUでよく使うコンパイルオプション一覧

タイプ	言語	Intel オプション	GNU オプション	内容
共通	共通	-qopenmp	-fopenmp	OpenMPを有効化
デバッグ オプション	共通	-g		実行ファイルにソースコードの情報を付与
		-Wall		コンパイル時にバグの元になりそうな箇所を検知
		-traceback	-fbacktrace	実行時にエラー発生個所を特定
	Fortran	-check bounds	-fbounds-check	実行中に初期化していない値へのアクセスなどを検知
最適化 オプション	共通	-O0、-O1、-O2、-O3		数字が大きいほど積極的な最適化を適用
		-xHost	-march=native	CPUアーキテクチャを考慮した最適化を有効化

# スパコンでのジョブの実行

OBCX スパコンでは、/home からプログラム実行できません  
→ **/work** ディレクトリ領域内にプログラム等一式をコピー

```
[tUVXYZ@obcx01 ~]$ cd   
[tUVXYZ@obcx01 ~]$ ls   
fibonacci  
[tUVXYZ@obcx01 ~]$ cp -r fibonacci /work/gt00/tUVXYZ/   
[tUVXYZ@obcx01 ~]$ cd /work/gt00/tUVXYZ/fibonacci 
```

作成したジョブスクリプトを投入、  
計算ノードで実行してもらいます。

```
[tUVXYZ@obcx01 fibonacci]$ pjsub fibonacci.sh 
```

## ジョブの確認、結果の表示

実行されているジョブを確認します。

(一瞬で計算が終わるので、ジョブ一覧に出ないかも)

```
[tUVXYZ@obcx01 fibonacci]$ pjstat
```

得られた結果を表示します。

```
[tUVXYZ@obcx01 fibonacci]$ more result_fibo.txt
```

[ **Return** ] で下の方に進めていくことができます]

結果から **747031** という文字列を探します

```
[tUVXYZ@obcx01 fibonacci]$ grep 747031 result_fibo.txt
```

## ジョブの管理コマンド

コマンド	内容	使い方
pjsub	ジョブの投入	\$pjsub “script.sh”
pjdel	ジョブの削除	\$pjdel “job ID”
pjstat	ジョブの状態	\$pjstat

注\* これらのコマンドはスパコンによって異なる

### pjstatのオプション

- -H : 終了したジョブの確認
- --rsc -b : 各リソースグループの混雑具合を確認可能
- --rsc -x : 各リソースグループで要求可能なリソース量を出力
- --nodeuse: OBCX全体での使用状況確認

# お試しアカウントについての注意

今回発行されたアカウント

→ 講習会終了後も約1ヶ月間、OBCXスパコンを使用可能

講習会終了後は リソースグループ lecture を使用してください。

リソースグループ tutorial は本日 13:00-17:00 のみです。

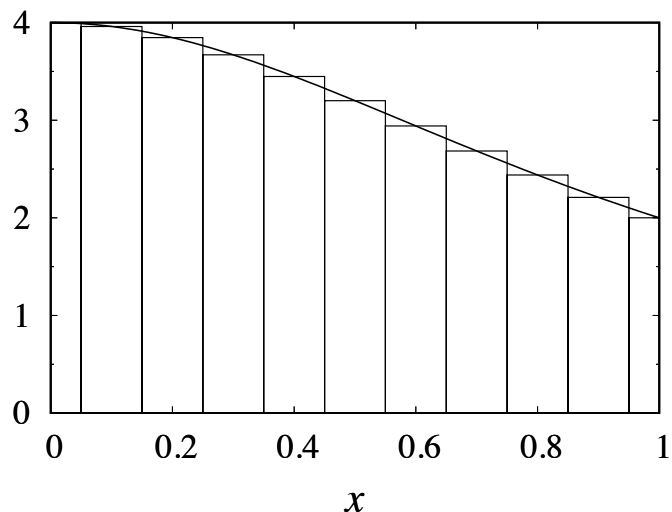
ジョブスクリプト  
fibonacci.sh

```
#!/bin/bash
#PJM -L rscgrp=lecture      [リソースグループ]
#PJM -L node=1
#PJM -L elapse=0:01:00
#PJM -g gt00
#PJM -N fibonacci
#PJM -o stdout.txt
#PJM -j

module purge
module load intel/2020.1.217
./fibonacci.out
```

# 並列計算の例題：区分解積分で円周率計算

$$I = \int_0^1 \frac{4}{1+x^2} dx$$
$$= \pi$$



和の短冊  
↓  
1億倍以上  
細かくする

数値積分を並列化 → 性能が向上する。

C, Fortran, Python, 3通りのサンプルプログラムを用意しました。

**実習：プログラムの並列度を変えながら実行  
実行時間から高速化の度合いを見る**



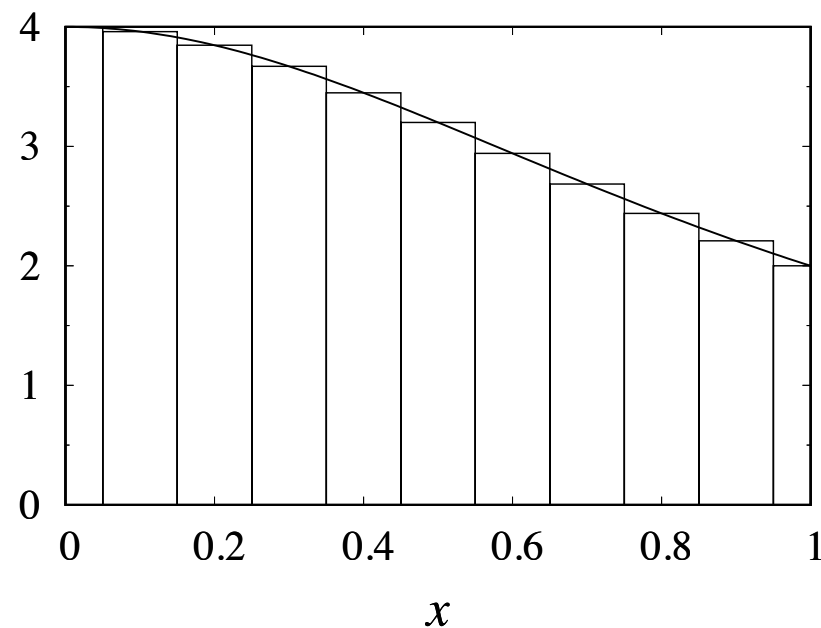
# 並列計算の例題：区分解積分で円周率計算

$$I = \int_0^1 \frac{4}{1+x^2} dx$$

変数変換  $x = \tan \theta$

$$dx = \frac{d\theta}{\cos^2 \theta}$$

$$\Rightarrow I = \int_0^{\pi/4} \frac{4}{1+\tan^2 \theta} \frac{d\theta}{\cos^2 \theta} = 4 \int_0^{\pi/4} d\theta = \pi$$



# コンピュータにおける演算の仕組み

(ノイマン型) コンピューター上では、全ての数がビット [0,1] の集まりで表現される

単精度浮動小数点 = 32ビットで1つの実数を表現

符号部 (sign) 1 bit      仮数部 (fraction), “b” 23 bit



指数部 (exponent), “e” 8 bit

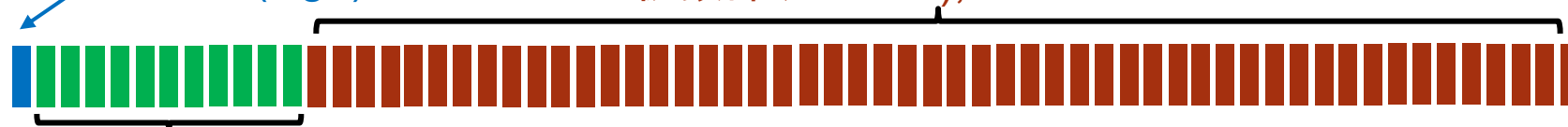
$$(-1)^{\text{sign}}(1.b_1b_2 \dots b_{23})_2 \times 2^{e-127}$$

$\log_{10}(2^{23}) \sim 7$  桁程度 (10進) が表現可

変数型	FORTRAN: real(kind=4)
	C, C++: float
	Python: 基本はなし

倍精度浮動小数点 = 64ビットで1つの実数を表現

符号部 (sign) 1 bit      仮数部 (fraction), “b” 52 bit



指数部 (exponent), “e” 11 bit

$$(-1)^{\text{sign}}(1.b_1b_2 \dots b_{52})_2 \times 2^{e-1023}$$

$\log_{10}(2^{52}) \sim 15$  桁程度 (10進) が表現可

変数型	FORTRAN: real(kind=8)
	C, C++: double
	Python: 無指定で倍精度

# プログラムの解説 C

## pi.c [非並列版]

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int i;
    int ndiv = 10000000;

    double width = 1.0 / (double)ndiv;
    printf("width = %.15f\n", width);
    double sum = 0.0;
    double x = 0.0;

    for (i=0; i<ndiv; i++) {
        x = (i+0.5)*width;
        sum += width * 4.0 / (1.0 + x*x);
    }

    printf("PI = %.15f\n", sum);
    return 0;
}
```

## pi\_mpi.c [並列版]

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
    int ndiv = 5600000000;
    int ierr, myrank, nprocs;
    int ndiv_local, i;
    double x, width, sum, total_sum;
    double t1, t2;

    width = 1.0 / (double)ndiv;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD,
                        &myrank);
    ierr = MPI_Comm_size(MPI_COMM_WORLD,
                        &nprocs);

    t1 = MPI_Wtime();

    sum = 0.0;
    ndiv_local = ndiv / nprocs;

    for (i = myrank*ndiv_local;
         i < (myrank+1)*ndiv_local; i++) {
        x = (i + 0.5) * width;
        sum = sum + width * 4.0 / (1.0 + x*x);
    }

    MPI_Reduce(&sum, &total_sum, 1, MPI_DOUBLE,
              MPI_SUM, 0, MPI_COMM_WORLD);

    t2 = MPI_Wtime();

    if (myrank == 0) {
        printf("PI(MPI) = %.18f\n", total_sum);
        printf("Number of cores utilized = %d\n", nprocs);
        printf("Execution time = %.8f (sec.)\n", t2 - t1);
    }

    ierr = MPI_Finalize();

    return 0;
}
```

## コンパイル

```
[tUVXYZ@obcx01 calc_pi_mpi]$ mpiicc pi_mpi.c -o pi_mpi.out
```

# プログラムの解説 Fortran

## pi.f90 [非並列版]

```

program main

implicit none

integer i, ndiv
real(kind=8) unit, width, sum, x

ndiv = 560000000
width = 1.0 / dble(ndiv)

print *, "width"
print '(F18.14)', width

sum = 0.0d0
x = 0.0d0

do i = 1, ndiv
  x = ( dble(i-1) + 0.5) * width
  sum += width * 4.0 / (1.0 + x*x)
end do

print *, "sum"
print '(F18.14)', sum

end program main

```

## pi\_mpi.f90 [並列版]

```

program main

Use mpi
implicit none

integer ndiv, ierr, myrank, nprocs
integer ndiv_local, i;
real(kind=8) unit, width, sum, x, total_sum
real(kind=8) t1, t2

ndiv = 5600000000
width = 1.0 / dble(ndiv)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,
                   myrank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,
                   nprocs, ierr)

sum = 0.0d0
ndiv_local = ndiv / nprocs

t1 = MPI_Wtime()

do i=myrank*ndiv_local+1, (myrank+1)*ndiv_local
  x = ( dble(i-1) + 0.5)*width
  sum = sum + width * 4.0 / (1.0 + x*x)
end do

call MPI_REDUCE(sum, total_sum, 1, MPI_REAL8,
MPI_SUM, 0, MPI_COMM_WORLD, ierr)

t2 = MPI_Wtime()

if (myrank .eq. 0) then
  print "(PI(MPI) = ', F18.16)", total_sum
  print "(Number of cores utilized = ', i0)", nprocs
  print "(Execution time = ', F12.8)", t2 - t1
endif

call MPI_FINALIZE(ierr)

end program main

```

## コンパイル

```
[tUVXYZ@obcx01 calc_pi_fortran]$ mpiifort pi_mpi.f90 -o pi_mpi.out
```

# プログラムの解説 Python

## pi.py [非並列版]

```
program main
implicit none

integer i, ndiv
real(kind=8) unit, width, sum, x

ndiv = 560000000
width = 1.0 / dble(ndiv)

print *, "width"
print '(F18.14)', width

sum = 0.0
x = 0.0

do i = 1, ndiv
  x = ( dble(i-1) + 0.5) * width
  sum = sum + width * 4.0 / (1.0 + x*x)
end do

print *, "sum"
print '(F18.14)', sum

end program main
```

## pi\_mpi.py [並列版]

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
myrank = comm.Get_rank();
nprocs = comm.Get_size();

ndiv = 560000000
width = 1.0/ndiv

t1 = MPI.Wtime()

sum = numpy.zeros(1)
total_sum = numpy.zeros(1)
ndiv_local = ndiv // nprocs

for i in range (myrank*ndiv_local, (myrank+1)*ndiv_local):
  x = width * (i+0.5)
  sum[0] = sum[0] + width * 4.0 / (1.0 + x*x)

comm.Reduce([sum, MPI.DOUBLE], total_sum, op=MPI.SUM, root=0)

t2 = MPI.Wtime()

if comm.rank == 0:
  print("PI(MPI) = ", total_sum[0])
  print("Number of cores utilized = ", nprocs)
  print("Execution time = ", t2 - t1, " (sec.)")
```

# プログラムなどを自分の /work へコピー

```
[tUVXYZ@obcx01 ~]$ cd /work/gt00/tUVXYZ
[tUVXYZ@obcx01 tUVXYZ]$ mkdir calc_pi_mpi
[tUVXYZ@obcx01 tUVXYZ]$ cd calc_pi_mpi
[tUVXYZ@obcx01 calc_pi_mpi]$ pwd
/work/gt00/tUVXYZ/calc_pi_mpi
```

## Fortran の方

```
$ cp /work/gt00/share/z30122/pi_fortran_mpi/* .
```

## C の方

```
$ cp /work/gt00/share/z30122/pi_c_mpi/* .
```

アスタリスク = ワイルドカード  
「ここはなんでも良い」

## Python の方

```
$ cp /work/gt00/share/z30122/pi_python_mpi/* .
```

# 並列プログラムのコンパイル

C および Fortran ではプログラムをコンパイルします。

C の方

```
[tUVXYZ@obcx01 calc_pi_mpi]$ mpiicc pi_mpi.c -o pi_mpi.out
```

Fortran の方

```
[tUVXYZ@obcx01 calc_pi_mpi]$ mpiifort pi_mpi.f90 -o pi_mpi.out
```



# Python の方

事前に次の作業が必要です。

```
[tUVXYZ@obcx01 ~]$ cd /work/gt00/tUVXYZ/calc_pi_mpi  
[tUVXYZ@obcx01 calc_pi_mpi]$ emacs setenv.sh (or vim setenv.sh)
```

次のところを変更してください

```
export PYTHONUSERBASE=/work/gt00/tUVXYZ/.local
```

↓

自分のユーザー名に

次にsetup.sh スクリプトを実行 (numpy, mpi4py をインストールします)

```
[tUVXYZ@obcx01 ~]$ ./setup.sh
```

# 並列プログラムのジョブスクリプト

性能が向上することを見ます。  
様々な並列度のジョブスクリプトを用意しました。

```
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n1c0004.sh - 1ノード, 4コア  
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n1c0028.sh - 1ノード, 28コア  
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n1c0056.sh - 1ノード, 56コア  
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n2c0112.sh - 2ノード, 112コア  
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n4c0224.sh - 4ノード, 224コア  
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n8c0448.sh - 8ノード, 448コア
```

## ジョブの実行状況の確認

自分の全てのジョブが終わるまで、待機します。

```
[tUVXYZ@obcx01 calc_pi_mpi]$ pjstat
```

ジョブが終了すると、出力ファイルが出ているはずです。

```
[tUVXYZ@obcx01 calc_pi_mpi]$ ls
```

pi_mpi.c	result_n2c0112.txt	run_n1c0056.sh
pi_mpi.out	result_n4c0224.txt	run_n2c0112.sh
result_n1c0004.txt	result_n4c0448.txt	run_n4c0224.sh
result_n1c0028.txt	run_n1c0004.sh	run_n8c0448.sh
result_n1c0056.txt	run_n1c0028.sh	

“**result\_n\*c\*\*\*\*.txt**” が、結果を記載した出力ファイルです。

# 結果を確認、実行時間を比較してみましょう

出力ファイルには実行時間の記載があります。  
出力ファイルの冒頭部分を一斉に見てみましょう。

```
[tUVXYZ@obcx01 calc_pi_mpi]$ head result*.txt
```

並列化で実行時間が短縮されています。

```
==> result_n1c0004.txt <==  
PI(MPI) = 3.141592653589913464  
Number of cores utilized = 4  
Execution time = **.***** (sec.)
```

```
==> result_n1c0028.txt <==  
PI(MPI) = 3.141592653589770912  
Number of cores utilized = 28  
Execution time = **.***** (sec.)
```

```
==> result_n1c0056.txt <==  
PI(MPI) = 3.141592653589800221  
Number of cores utilized = 56  
Execution time = **.***** (sec.)
```

```
==> result_n2c0112.txt <==  
PI(MPI) = 3.141592653589794892  
Number of cores utilized = 112  
Execution time = **.***** (sec.)
```

```
==> result_n4c0224.txt <==  
PI(MPI) = 3.141592653589791340  
Number of cores utilized = 224  
Execution time = **.***** (sec.)
```

```
==> result_n8c0448.txt <==  
PI(MPI) = 3.141592653589797557  
Number of cores utilized = 448  
Execution time = **.***** (sec.)
```

## 参考情報：OBCXのリソースグループ

### リソースグループ毎のノード数および最長実行時間一覧

リソースグループ名	ノード数	最長実行時間	説明
debug	1～16	30分	デバッグ用
short	1～8	8時間	短いジョブ用
regular	1～128	48時間	通常使うリソースグループ
	129～256	24時間	
interactive	1	2時間	インタラクティブジョブ用（後述）
	2～8	10分	
tutorial	1～8	15分	講習会用
lecture	1～8	15分	講習会後一か月間有効

- 講習会中はtutorialをご使用ください。
- 講習会後から利用期限まではlectureをご利用ください。
- それ以外のリソースグループは使用できません。

# 参考情報：計算ノードで直接操作したい場合

通常、スパコンでは  
ログインノードと呼ば  
れる玄関口から実行  
命令を出します



端末

1. ログイン  
SSH



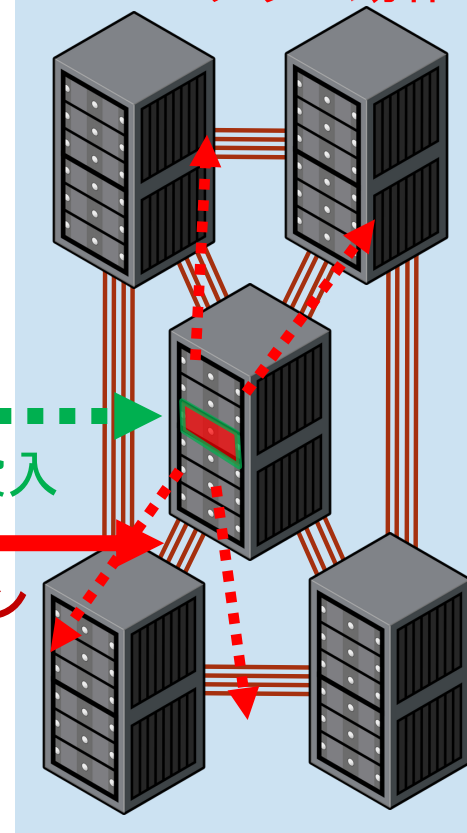
ログインノード

```
[tut138@obcx02~]$ ls -l  
drwxr-x--- 2 shiba group 10 1  
Apr 13:00 test.out  
[tut138@obcx02~]$ ./test.out  
Hello world  
[tut138@obcx02~]$ qsub a.sh
```

2. ジョブ投入

2'. ログイン

3. プログラム動作



計算ノード  
= 本体

## 参考情報：計算ノードで直接操作したい場合

インタラクティブジョブ  
対話型のジョブ デバッグなどに便利

手元で実行しているのと同様の挙動

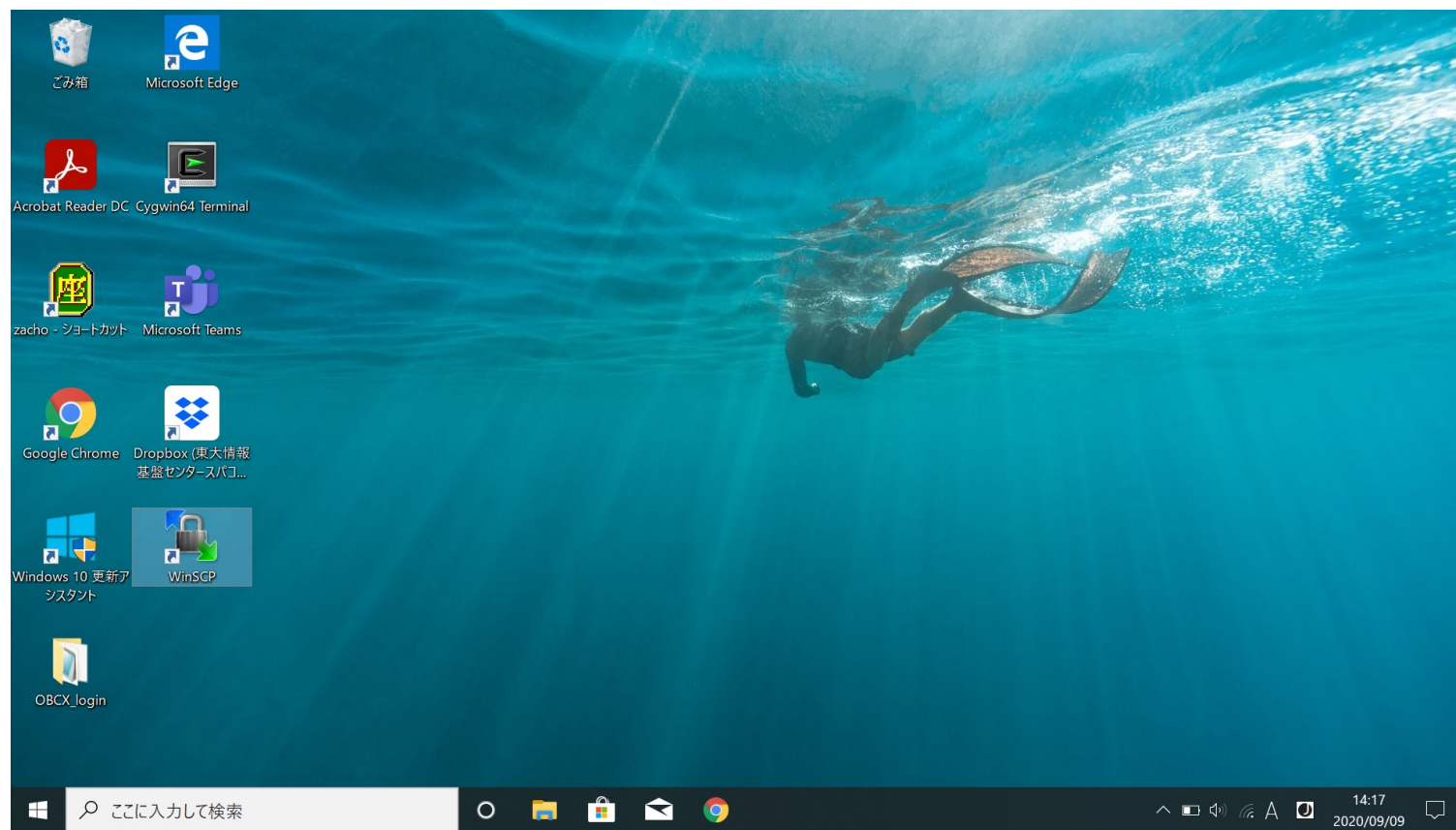
```
[tUVXYZ@obcx04 tUVXYZ]$ psub --interact -g gt00 -L rscgrp=interactive,node=1  
[INFO] PJM 0000 psub Job 517079 submitted.  
[INFO] PJM 0081 .connected.  
[INFO] PJM 0082 psub Interactive job 517079 started.  
[tUVXYZ@cx0065 tUVXYZ]$
```

ログインノード(obcx04)から計算ノード(cx0065)へログインしたような状態で  
プログラムの実行が可能

ログインノードとは違い専用の領域にいるので、負荷のある実行をして良い

# 最後に：計算結果のファイルを手元に転送する

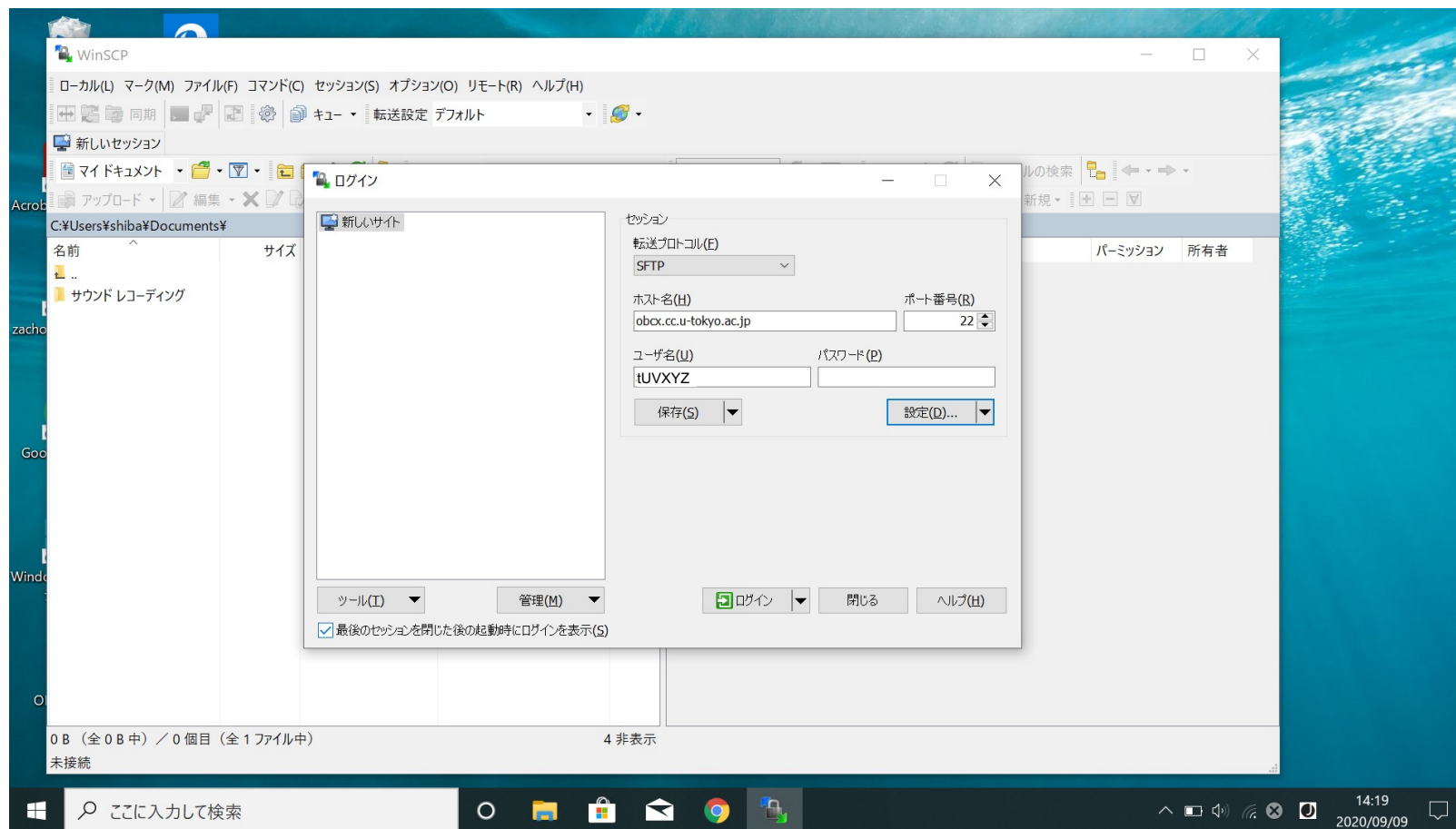
WinSCP を起動してください





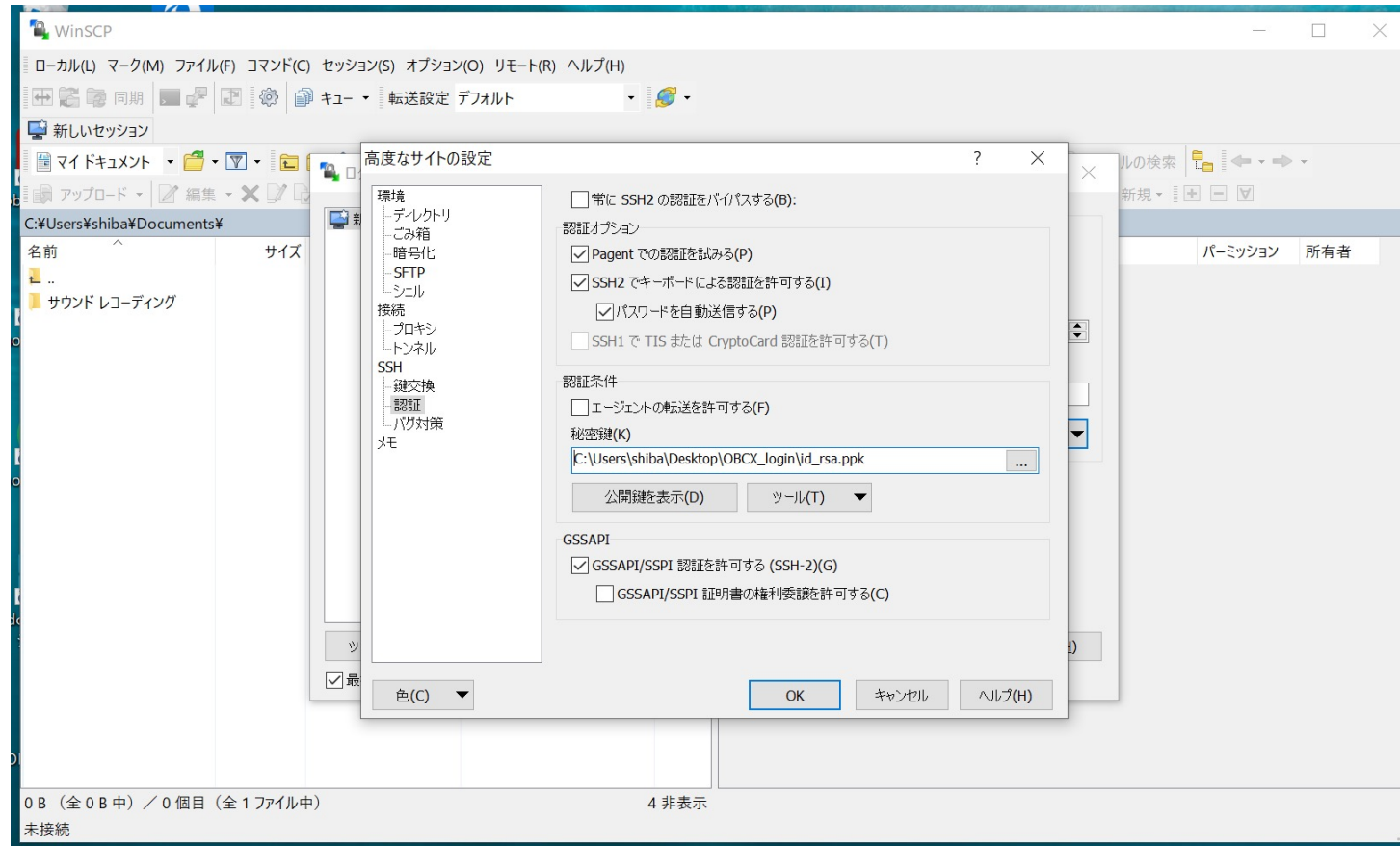
# 最後に：計算結果のファイルを手元に転送する

接続サーバー情報を、ユーザー名 (tUVXYZ) とともに入力します  
OBCXでは鍵認証ですので、**パスワードは空欄のまま**



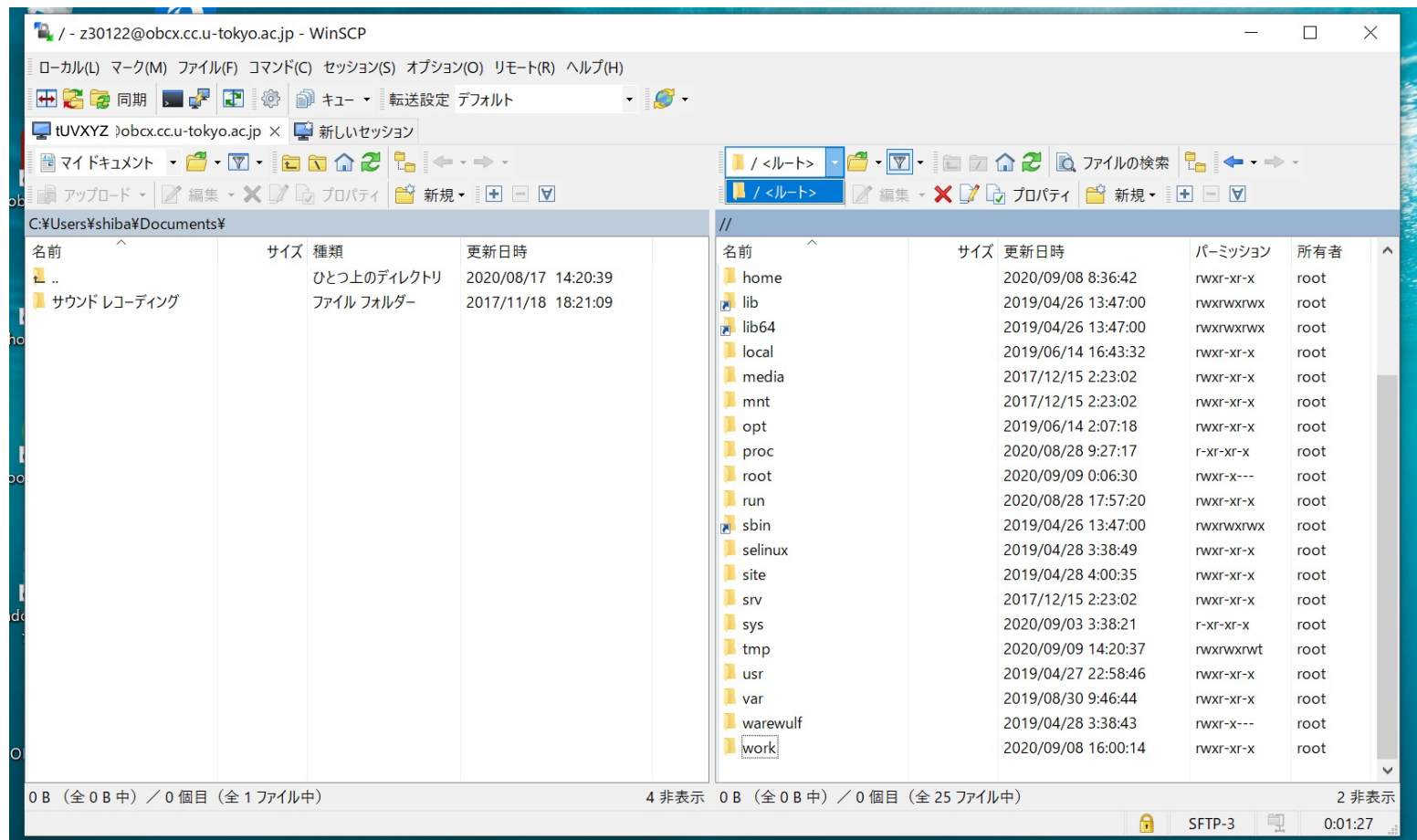
# 最後に：計算結果のファイルを手元に転送する

SSH > 認証 秘密鍵を指定し、チェックボックスを確認



# 最後に：計算結果のファイルを手元に転送する

目的のディレクトリに行くために、まず「 / <ルート>」から出発



# 最後に：計算結果のファイルを手元に転送する

こんな画面になります

